Heuristic Generation of Software Test Data

Stephen Terry Holmes

A thesis submitted in partial fulfilment of the requirements of the University of Glamorgan for the degree of Doctor of Philosophy

September 1996

Department of Computer Studies
The University of Glamorgan
Pontypridd
Mid Glamorgan
CF37 1DL

Abstract

Incorrect system operation can, at worst, be life threatening or financially devastating. Software testing is a destructive process that aims to reveal software faults. Selection of good test data can be extremely difficult. To ease and assist test data selection, several test data generators have emerged that use a diverse range of approaches. Adaptive test data generators use existing test data to produce further effective test data. It has been observed that there is little empirical data on the adaptive approach.

This thesis presents the Heuristically Aided Testing System (HATS), which is an adaptive test data generator that uses several heuristics. A heuristic embodies a test data generation technique. Four heuristics have been developed. The first heuristic, Direct Assignment, generates test data for conditions involving an input variable and a constant. The Alternating Variable heuristic determines a promising direction to modify input variables, then takes ever increasing steps in this direction. The Linear Predictor heuristic performs linear extrapolations on input variables. The final heuristic, Boundary Follower, uses input domain boundaries as a guide to locate hard-to-find solutions. Several Ada procedures have been tested with HATS; a quadratic equation solver, a triangle classifier, a remainder calculator and a linear search. Collectively they present some common and rare test data generation problems.

The weakest testing criterion HATS has attempted to satisfy is all branches. Stronger, mutation-based criteria have been used on two of the procedures. HATS has achieved complete branch coverage on each procedure, except where there is a higher level of control flow complexity combined with non-linear input variables. Both branch and mutation testing criteria have enabled a better understanding of the test data generation problems and contributed to the evolution of heuristics and the development of new heuristics.

This thesis contributes the following to knowledge :

- Empirical data on the adaptive heuristic approach to test data generation.
- How input domain boundaries can be used as guidance for a heuristic.
- An effective heuristic termination technique based on the heuristic's progress.
- A comparison of HATS with random testing. Properties of the test software that indicate when HATS will take less effort than random testing are identified.

Contents

Abstract .... *ii*

Contents .... *iii*

List of Tables .... *viii*

List of Figures .... *xi*

Acknowledgements .... *xiii*

Glossary .... *xiv*

Certificate of Research .... *xvi*

Declaration .... *xvii*

1 Introduction *1*

2 Automatic Test Data Generation *3*

2.1 Introduction **3**

2.2 Fundamental Concepts **3**

2.3 Testing Methods **4**

2.4 Test Data Selection **5**

2.5 Automatic Test Data Generators **6**

2.5.1 Random Test Data Generation 6
2.5.2 Syntax Based Test Data Generation 6
2.5.3 Specification Based Test Data Generation 6
2.5.4 Test Data Generation by Symbolic Execution and Solution of Produced Constraints 6
2.5.5 Adaptive Test Data Generation 7

3 The Heuristically Aided Testing System *11*

3.1 Introduction **11**

3.2 Objectives of HATS **11**

3.3 The Closeness Phenomenon and its Exploitation **11**

3.4 HATS Architecture **12**

3.5 Testing Criterion, Software Model and Instrumentation **13**

3.5.1 Testing Criterion 13
3.5.2 Software Model 13
3.5.3 Instrumentation 15
3.5.4 Selecting Branches to Test 16
3.5.5 The Input Point Closeness Metric 16
3.5.6 Adequacy Criterion Influence on the Software Model and Instrumentation 17
3.5.7 Testing at Various Levels of Abstraction 18
3.5.8 Reuse of the Software Model and its Data 18

3.6 The HATS Harness I8

3.7 Heuristics 20

3.8 Selection of Initial Points 21

3.9 The Direct Assignment Heuristic 21
3.9.1 Overview of the Direct Assignment Heuristic 22
3.9.2 Components and Functions of the Direct Assignment Heuristic 22

3.10 The Alternating Variable Heuristic 23
3.10.1 Overview and Phases of the Alternating Variable Heuristic 23
3.10.2 Components and Function of the Alternating Variable 25

3.11 Scope of Testable Software 28
3.11.1 Testable Subset of Ada 28
3.11.2 Transformations 29

3.12 Automated and Manual Aspects of HATS 29

4 The Quadratic Equation Solver Problem *30*

4.1 Introduction 30

4.2 The Integer Quadratic Equation Solver Procedure 31
4.2.1 The QUADRATIC's Input Space 31

4.3 Branch Testing of the QUADRATIC 33
4.3.1 HATS Experimental Set-up 33
4.3.2 Run Q1 33
4.3.3 Run Q2 37
4.3.4 Run Q3 38
4.3.5 Run Q4 39
4.3.6 Branch Testing Discussion 39
4.3.7 Comparison of HATS with Random Testing 40

4.4 Mutation Analysis 41
4.4.1 The Appeal of Mutation Analysis 41
4.4.2 Using Mutation Analysis with HATS 42
4.4.3 Mutants Produced 43

4.5 The First Round of Mutation Analysis 44
4.5.1 Experimental Set-up 44
4.5.2 Statement Analysis Results and Discussion 44
4.5.3 Predicate Analysis Results and Discussion 46
4.5.4 Mutation Analysis Summary 47

4.6 Heuristic Discussion 48
4.6.1 DA Discussion 48
4.6.2 AV Discussion 48

5 The New Heuristics and Improved Quadratic Results *50*

5.1 Introduction 50

5.2 The Linear Predictor Heuristic 50
5.2.1 Overview and Phases of the Linear Predictor 50
5.2.2 Components and Functions of the Linear Predictor 51

5.3 The Boundary Follower Heuristic....59
5.3.1 Following Domain Boundaries in the Program Input Space....60
5.3.2 Overview of the Boundary Follower....60
5.3.3 The Follow Role....61
5.3.4 The Cross Role....61
5.3.5 Following a Boundary Using Follow and Cross Moves....62
5.3.6 Establishing the Correct Cross Move Direction....63
5.3.7 Reorienting the Boundary Follower....64
5.3.8 Locating a Point to Commence Boundary Following From....66
5.3.9 Initial Allocation of the Follow and Cross Roles....66
5.3.10 Phases of the Boundary Follower Heuristic....68
5.3.11 Components and Functions of the Boundary Follower....68
5.3.12 Known Limitations....72

5.4 Installing the LP and BF into the HATS Harness....73

5.5 The Second Round of Mutation Analysis....73
5.5.1 Experimental Set-up....73
5.5.2 Statement Analysis Results....74
5.5.3 Predicate Analysis Results....75
5.5.4 Improving Upon the AV....75

5.6 The Third Round of Mutation Analysis....76
5.6.1 Experimental Set-up....76
5.6.2 Results....76

5.7 DA Improvements....77

5.8 Mutation Analysis Discussion and Conclusions....77

6 The Triangle Classification Problem 78

6.1 Introduction....78

6.2 Experimental Conditions....78

6.3 TRIANGLE Experiments....78
6.3.1 Run T1....80
6.3.2 Run T2....81
6.3.3 Run T3....83
6.3.4 Comparison of HATS with Random Testing on the TRIANGLE Procedure....86

6.4 TRIANGLE_2 Experiments....86
6.4.1 Run T4....88
6.4.2 Run T5....89
6.4.3 Run T6....89
6.4.4 Comparison of HATS with Random Testing on the TRIANGLE_2 Procedure....89

6.5 RIGHT_ANGLE_CHECK Experiments....90
6.5.1 Run T7....92
6.5.2 Run T8....93
6.5.3 Run T9....95
6.5.4 Comparison of HATS with Random Testing on the RIGHT_ANGLE_CHECK Procedure....96

6.6 TRIANGLE_COMPLETE Experiments....................................97
6.6.1 Run T10..100
6.6.2 Run T11..102
6.6.3 Run T12..103
6.6.4 Comparison of HATS with Random Testing on the TRIANGLE_COMPLETE Procedure.......................................106

6.7 Overall Discussion..107
6.7.1 Direct Assignment Heuristic...108
6.7.2 Linear Predictor Heuristic...108
6.7.3 Boundary Follower Heuristic...108

7 The Remainder and Linear Search Problems...........................109

7.1 Introduction..109

7.2 The REMAINDER Procedure..109

7.3 Techniques for Testing Loops...111

7.4 REMAINDER Branch Testing Experiments..........................113
7.4.1 HATS Experimental Set Up...113
7.4.2 Run R1..113
7.4.3 Run R2..114
7.4.4 Run R3..114
7.4.5 Run R4..115
7.4.6 Run R5..116
7.4.7 Overall Discussion for the HATS Branch Testing Runs........117
7.4.8 Comparison of HATS with Random Testing on the REMAINDER Procedure..118

7.5 Mutation Analysis of the REMAINDER Procedure..................119
7.5.1 Mutation Experiments...119
7.5.2 Mutation Analysis Results and Discussion........................121

7.6 The LINEAR_SEARCH Procedure..122

7.7 Techniques for Testing Arrays..123

7.8 LINEAR_SEARCH Branch Testing Experiments......................124
7.8.1 HATS Experimental Set Up...124
7.8.2 Run LS1...124
7.8.3 Run LS2...125
7.8.4 Run LS3...125
7.8.5 Run LS4...125
7.8.6 HATS Discussion..126
7.8.7 Comparison of HATS with Random Testing on the LINEAR_SEARCH Procedure...126

8 Conclusions and Further Work..127

8.1 Introduction..127

8.2 Conclusions..127
8.2.1 HATS is an Improvement Over Random Testing..................127
8.2.2 Domain Boundaries Can be Used as a Guide to Solution Points............128

8.2.3 Control Deviations From Partial Paths to a Considered Node Have a Detrimental Affect on Heuristic Performance 128
8.2.4 Termination Criteria Based on Promising Effects are Effective and Efficient 128
8.2.5 Coincidental Traversal Can be Considerable 128
8.2.6 The Initial Point Set Influences the Effort and Success of the Heuristics 129
8.2.7 Point Metrics Based on the Closeness Phenomenon have Limitations.... 129
8.2.8 Heuristic Based Test Data Generation is a Promising Approach........ 130

8.3 Further Work........ 130
8.3.1 HATS 130
8.3.2 Heuristics........ 132
8.3.3 Alternative Representations of the Test Software as a Test Data Generation Problem 133
8.3.4 Metrics........ 134
8.3.5 Scope of Testable Software........ 135
8.3.6 Input Space Study........ 135
8.3.7 New Tools 136

Appendix 1 HATS User Summary 137

Appendix 2 Quadratic Mutation Testing Initial Points........ 139

Appendix 3 Heuristics' Pseudo Code 140

A3.1 Direct Assignment Heuristic........ 140
A3.1.1 First Iteration Set-up 140
A3.1.2 Generator........ 140
A3.1.3 Duplicate Data Handler 140
A3.1.4 Evaluator........ 140

A3.2 Alternating Variable Heuristic........ 141
A3.2.1 First Iteration Set-up 141
A3.2.2 Generator........ 141
A3.2.3 Duplicate Data Handler 141
A3.2.4 Evaluator........ 142

A3.3 Linear Predictor Heuristic 143
A3.3.1 First Iteration Set-up 143
A3.3.2 Generator........ 144
A3.3.3 Evaluator........ 144

A3.4 Boundary Follower Heuristic 147
A3.4.1 First Iteration Set-up 147
A3.4.2 Generator........ 147
A3.4.3 Evaluator........ 148

References 153

List of Tables

3.1 DA's generation instructions for the considered predicate's input variable, defined by the considered predicate's relational operator 23
3.2 Determining exploratory phase input point generation instructions............ 25
4.1 The QUADRATIC's input domains 32
4.2 Initial points for branch testing of the QUADRATIC 33
4.3 Run Q1 traversal results 34
4.4 Run Q1's iteration summary for the nodes considered 34
4.5 Progression of the AV considering node 6 36
4.6 Run Q2 traversal results 37
4.7 Progression of the AV considering node 6 38
4.8 Run Q3 traversal results 38
4.9 Progression of the AV considering node 6 38
4.10 Run Q4 traversal results 39
4.11 Progression of the AV considering node 6 39
4.12 Relationship between the predicate value distance to an expected solution and the number of iterations taken to locate the solution 40
4.13 The number of iterations taken by HATS and random testing for each of the QUADRATIC's nodes 41
4.14 QUADRATIC's first round statement analysis mutants 43
4.15 QUADRATIC's first round predicate analysis mutants 44
4.16 First round statement analysis mutants revealed by the initial point 45
4.17 First round statement analysis mutants revealed by a heuristic 45
4.18 First round statement analysis mutants unrevealed by the initial point or heuristics 46
4.19 First round predicate analysis mutants revealed by the initial point 46
4.20 First round predicate analysis mutants revealed by a heuristic 47
4.21 First round predicate analysis mutants unrevealed by the initial point or heuristics 47
4.22 Overall mutation analysis results for the QUADRATIC 48
5.1 HATS run excerpt : LP in the Determine-linearity phase considering node 5 of the QUADRATIC and modifying input variable A 52
5.2 HATS run excerpt : LP in the Predictor phase considering node 5 of the QUADRATIC and modifying input variable A 54
5.3 HATS run excerpt of the LP in the Determine-linearity phase considering node 6 of the QUADRATIC, demonstrating the use of the unpromising effects count to minimise unnecessary iterations 58
5.4 Events that cause the unpromising effects count to be incremented by one..... 59
5.5 Events that cause the unpromising effects count to be reset to zero 59
5.6 Follow and Cross role suitability when the Central point's predicate value is greater than the boundary's predicate value 67
5.7 Follow and Cross role suitability when the Central point's predicate value is less than the boundary's predicate value 67
5.8 Progress to the final allocation of roles in the Determine-initial-follow-and-cross-details phase with the QUADRATIC 70
5.9 HATS run excerpt : BF in the Follow-boundary phase considering node 6 of the QUADRATIC and locating a solution point 71
5.10 Second round statement analysis mutants revealed by the BF on node 6 74

5.11 Second round statement analysis mutants unrevealed by the BF on node 6 in 500 iterations 74
5.12 Second round predicate analysis mutants revealed by the BF on node 6 75
5.13 Second round predicate analysis mutants unrevealed by BF on node 6 in 500 iterations 75
5.14 Iterations taken by the AV and BF on node 4 or 5 of the QUADRATIC 76
5.15 Third round statement analysis mutants revealed by the BF on node 6 77
6.1 TRIANGLE procedure's initial points 79
6.2 Run T1 traversal results 80
6.3 Run T1 excerpt : LP considering node 10 in the Determine-linearity phase 81
6.4 Run T1 excerpt : LP considering node 12 in the Determine-linearity phase 81
6.5 Run T2 traversal results 81
6.6 Run T2 excerpt : LP considering node 10 in the Determine-linearity phase modifying variable A 82
6.7 Run T2 excerpt : LP considering node 10 in the Determine-linearity phase identifying variable B as linear 82
6.8 Run T2 excerpt : LP considering node 10 in the Predicted-point phase, causing just sibling traversal then considered node traversal, modifying variable B 82
6.9 Run T3 traversal results 83
6.10 Run T3 excerpt : LP considering node 12 in the Determine-linearity phase modifying variable A 83
6.11 Run T3 excerpt : LP considering node 12 in the Predicted-point phase, modifying variable A to locate a point that causes sibling-traversal after predicting a point and producing an upper-deviation 84
6.12 Run T3 excerpt : LP considering node 12 in the Determine-linearity phase having no success 84
6.13 Run T3 excerpt : BF considering node 12 in the Obtain-a-close-point phase, having no success 85
6.14 Run T3 excerpt : BF considering node 12 in the Obtain-a-close-point phase succeeding by modifying variable C 85
6.15 The number of iterations taken by HATS and random testing for each of TRIANGLE's nodes 86
6.16 TRIANGLE_2 procedure's initial points 88
6.17 Run T4 traversal results 88
6.18 Run T5 traversal results 89
6.19 Run T6 traversal results 89
6.20 The number of iterations taken by HATS and random testing for each of TRIANGLE_2's nodes 90
6.21 RIGHT_ANGLE_CHECK procedure's initial points 91
6.22 Run T7 traversal results 92
6.23 Run T7 excerpt : LP creeping to a solution point, modifying variable A, on non-linear node 2 92
6.24 Run T7 excerpt : BF finding a solution for non-linear node 6 92
6.25 Run T8 traversal results 93
6.26 Run T8 excerpt : BF's first reorientation considering node 6 94
6.27 Run T8 excerpt : BF's second reorientation considering node 6 95
6.28 Run T8 excerpt : BF's third reorientation considering node 6 95
6.29 Run T9 traversal results 96

6.30 The number of iterations taken by HATS and random testing for each of RIGHT_ANGLE_CHECK's nodes 96
6.31 TRIANGLE_COMPLETE procedure's initial points 100
6.32 Run T10 traversal results 100
6.33 Run T11 traversal results 102
6.34 Run T12 traversal results 104
6.35 Run T12 excerpt : BF considering node 23 in the first Follow-boundary phase 104
6.36 The number of iterations taken by HATS and random testing for each of TRIANGLE_COMPLETE's nodes 107
6.37 HATS branch coverage on the triangle related procedures 107
7.1 The 6 cases handled in the REMAINDER procedure 109
7.2 REMAINDER procedure's initial points 113
7.3 Run R1 traversal results 113
7.4 Run R2 traversal results 114
7.5 Run R3 traversal results 114
7.6 Run R4 traversal results 116
7.7 Run R5 traversal results 116
7.8 The number of iterations taken by HATS and random testing for each of REMAINDER's nodes 118
7.9 Mutants produced from the REMAINDER procedure using the mutation operator, predicate alteration by a small amount 120
7.10 Results from mutation testing the REMAINDER procedure 121
7.11 LINEAR_SEARCH's initial points and the order heuristics will consider input variables 124
7.12 Run LS1 excerpt : LP considering node 3, modifying variable X; array A stayed constant with elements 1 to 5 having values 1, 4, 3, 5 and 3 respectively 124
7.13 Run LS4 excerpt : LP considering node 3, modifying variable A(5) in the Determine-linearity phase; array A (1..4) and X stayed constant having values -93, -21, 82, -43 and 27 respectively 125
7.14 Run LS4 excerpt : LP considering node 3, modifying variable A(1); array A (2..5) and X stayed constant having values -21, 82, -43, 41 and 27 respectively 125
7.15 The number of iterations taken by HATS and random testing for each of the LINEAR_SEARCH's nodes 126
A2.1 Initial points for the first round of QUADRATIC mutation analysis 139

List of Figures

3.1 HATS Architecture 12

3.2 The TEST_ME procedure and its control flow tree 14

3.3 TEST_ME's control flow tree showing each traversed nodes' linked list........ 15

3.4 Procedure TEST_ME with instrumentation 16

3.5 Structure diagram of the HATS harness 19

3.6 Components and flow in a heuristic 21

3.7 Partial input plane showing the AV operating 24

3.8 Partial input plane showing moves made by the AV considering node 4 of the TEST_ME procedure 26

4.1 The Ada QUADRATIC procedure 30

4.2 Control flow tree of the QUADRATIC procedure 31

4.3 Partial planes of the QUADRATIC's input space 32

4.4 Approximated partial plane of the QUADRATIC's input space showing AV heuristic moves on node 5 35

4.5 Approximated partial plane of the QUADRATIC's input space showing the AV's moves on node 6........ 35

4.6 Run Q1 (A, B) partial input plane : AV considering node 6 36

4.7 Run Q1 (B, C) partial input plane : AV considering node 6........ 37

5.1 Operation of the Linear Predictor 51

5.2 (A, B) partial input plane : LP in the Determine-linearity phase considering node 5 of the QUADRATIC........ 52

5.3 (A, B) partial input plane : LP in the Predictor phase considering node 5 of the QUADRATIC 54

5.4 (A, B) partial input plane : LP in the Determine-linearity and Creeper phases, considering node 5 of the QUADRATIC 56

5.5 (A, B) and (B, C) partial input planes : LP in the Determine-linearity phase, considering node 6 of the QUADRATIC 58

5.6 Operation of the Boundary Follower 61

5.7 Partial input planes showing Follow and Cross moves, with the Follow role allocated to variable A and Cross to B 62

5.8 A domain boundary being followed 63

5.9 Use of the Cross rule to determine the Cross direction........ 64

5.10 A border change which renders the BF unable to cross the boundary with Cross variable A, and Follow variable B 65

5.11 Reorienting the BF........ 65

5.12 Continuation of boundary following subsequent to reorientation with Follow variable A, Cross variable B and Follow direction of increase........ 66

5.13 Modifications made to the QUADRATIC's input variables during the Determine-initial-follow-and-cross-details phase of the BF 69

5.14 (A, B) partial input plane showing Follow-boundary phase moves to a solution point for node 6 of the QUADRATIC 71

5.15 Partial input plane showing Follow-boundary phase moves, had variable A been allocated Follow and B allocated Cross, to a solution point for node 6 of the QUADRATIC........ 72

5.16 New heuristic selection order with the LP and BF heuristics added........ 73

6.1 The TRIANGLE procedure........ 79

6.2 Control flow tree of the TRIANGLE procedure 80

6.3 Run T2 (A, B) partial input plane : LP considering node 10........ 83

6.4 Run T3 (A, C) partial input plane : LP considering linear node 12 and failing. 84
6.5 Run T3 (B, C) partial input plane : BF considering node 12 and succeeding... 85
6.6 The TRIANGLE_2 procedure........ 87
6.7 Control flow tree of the TRIANGLE_2 procedure 87
6.8 The RIGHT_ANGLE_CHECK procedure 91
6.9 Control flow tree of the RIGHT_ANGLE_CHECK procedure........ 91
6.10 Run T7 (A, B) partial input plane : BF considering node 6........ 93
6.11 Run T8 (B, C) partial input plane : BF considering node 6 and reorienting three times 93
6.12 The TRIANGLE_COMPLETE procedure 98
6.13 Control flow tree of the TRIANGLE_COMPLETE procedure........ 99
6.14 Run T10 (A, B) partial input plane : BF considering node 23 in the Determine-initial-follow-and-cross-details phase........ 101
6.15 Run T10 (B, C) partial input plane : BF considering node 23 in the Determine-initial-follow-and-cross-details phase........ 101
6.16 Run T11 (A, B) partial input plane : LP considering node 17........ 103
6.17 Run T11 (B, C) partial input plane : LP considering node 17........ 103
6.18 Run T12 (B, C) partial input plane : BF considering node 23; from the heuristic's start point........ 105
6.19 Run T12 (B, C) partial input plane : BF considering node 23; showing part of the boundary followed and a solution point 106
7.1 The REMAINDER procedure........ 110
7.2 Control flow tree of the REMAINDER procedure........ 111
7.3 The modelling and instrumentation of a loop 112
7.4 Run R1 partial input space : LP considering node 15 113
7.5 Run R2 partial input space : LP considering node 19 114
7.6 Run R3 partial input space : LP and BF considering node 13 115
7.7 Run R3 partial input space : LP and BF considering node 17 115
7.8 REMAINDER procedure's input space........ 117
7.9 The LINEAR_SEARCH procedure........ 122
7.10 Control flow tree of the LINEAR_SEARCH procedure 122
7.11 Comparison of instrumentation for scalar variable and array variable conditions 123
A1.1 Flow diagram showing the HATS testing process........ 137

Acknowledgements

I am most grateful to both my supervisors, Mr. David Eyres and Professor Bryan Jones, for the guidance and support they have provided. I would like to thank both Professor Darrel Ince and Dr David Nibb for the perspective and polarisation they gave. Both my fellow researchers, Dr. Harmen Sthamer and Ms. Xila Yang, deserve thanks for the inspiration they provided.
Finally, to my wife and soul-mate Shima, thank you for all the encouragement and support you have given me. It is good to know that you are always there.

Glossary

The following list is of abbreviations are used throughout the thesis :

AV	Alternating Variable heuristic
BF	Boundary Follower heuristic
DA	Direct Assignment heuristic
DIFCD	Determine-initial-follow-and-cross-details (phase of BF)
DL	Determine-linearity (phase of LP or sub-phase of OCP in BF)
FB	Follow-boundary (phase of BF)
HATS	Heuristically Aided Testing System
iter(s)	iteration(s)
LP	Linear Predictor heuristic
MA	Mutation Analysis
OCP	Obtain-a-close-point (phase of BF)
RBF	Reorient-boundary-follower (phase of BF)
sib-trav	sibling-traversal

The following list is of abbreviations are used in HATS run excerpt headings :

FB	Follow-boundary
Pred	Predicate
Trav	Traversal
UP eff	Unpromising effects
Var	Variable

The following list is of abbreviations are used in the body of HATS run excerpts :

+(var)	Increase specified variable; e.g. +A
+ST	Positive sibling-traversal
-(var)	Decrease specified variable; e.g. -A
-Cr	Decrease Cross variable
-ST	Negative sibling-traversal
B BP	modify considered variable Back toward Base Point
BP	Base Point
C (val)	Cross move, modify cross variable, according to cross rule, by value
C +(val)	Cross move, increase cross variable by value; e.g. C +1
C -(val)	Cross move, decrease cross variable by value; e.g. C -2
F	Follow move
FB	Follow-boundary phase
FB C	Follow-boundary phase, make a Cross move
FB F	Follow-boundary phase, make a Follow move
NT	considered Node Traversal
O DL	Obtain-a-close-point : Determine-linearity
O PP	Obtain-a-close-point : Predicted-point
OF +C	Opposite direction to Follow and increase Cross variable
PP	Predicted Point
ST	Sibling-traversal
SUCC	Success - traversal of the considered node
TERM	Terminate
TFV	Try First Variable
TNV	Try Next Variable
UD	Upper-deviation

Certificate of Research

This is to certify that except where specific reference is made, the work presented in this thesis is the result of the investigation undertaken by the candidate.

Candidate SHolmes. ....

Director of Studies ....

Declaration

This is to certify that neither this thesis or any part of it has been presented or is being currently submitted in candidature for any degree other than the degree of Doctor of Philosophy at the University of Glamorgan.

Candidate S Holmes

1 Introduction

On the morning of the 15th March 1995 a new, ultra high-tech and therefore packed with software, Airbus A340 was on the final part of its approach to Gatwick airport. Both of the pilots screens had gone blank except for the message "Please wait....". Unnerved by this, the pilots requested that the plane turn left. The plane responded by turning right. Perhaps now a little concerned, they tried to get the plane to adopt a 3 degree approach to the runway. The plane responded by adopting a 9 degree plummet. By now the passengers as well as the pilots, may well have been on the verge of panic and worried for their lives. Fortunately however, the pilots managed to gain manual control and land safely.

This example, taken from a BBC news report, illustrates the responsibility society places in systems and the potential for disaster when the unexpected occurs. Risks from systems (Neuman, 1995) span the whole spectrum of impact, both direct and, even harder to assess, indirect. Even systems which exactly meet their specification still present some risk to society.

We depend upon systems. Many wonderful things are possible with their assistance and would be impossible or very difficult without. As the information era progresses our dependence will increase. Our need for quality software will correspondingly increase. However, as technology has improved over the past 30 years, the quality of software has not improved at a similar rate.

Various techniques are used to improve the quality of software. Software testing has been used since the first program was written. Software testing aims to reveal faults, which cause incorrect or unspecified behaviour. Unfortunately, testing in general, cannot guarantee the absence of all faults. Test data are selected toward satisfying an adequacy criterion. This criterion states when testing can stop and at that point there should be few faults left in the software. This should have established an acceptable degree of confidence in the software.

Over the last 20 years methods have been developed (Pressman, 1994) which structure the process of software development. By using a method, more faults are removed at each stage of software development. Test data can be selected using the products generated from each of the method's stages. A complementary approach to software testing is software inspections (Fagan, 1976) and walk-throughs (Myers, 1979). These reveal many faults without executing the test software.

Software testing is not a panacea and has a number of problems. It consumes significant resources and time. Myers (1979) states "in a typical programming project, approximately 50% of the elapsed time and over 50% of the total cost are expended in testing the program or system being developed". Graham (1991) states

"Testing typically takes 40% of development effort" and continues "Testing often uses far too much of the most expensive resource : people". A great deal of creativity must go into the selection of fault revealing test data. However, software testing is essentially a destructive and repetitive process, which can make it an unattractive activity.

These problems can be eased through automation of testing. Graham (1991) reports that quality improvements of 95 to 100% and productivity improvements of up to 7500% can be achieved with computer aided software testing tools. It is generally accepted that the most difficult aspect of software testing is test data selection. Ince (1987) outlines several approaches. Of these approaches he describes adaptive testing as a "tantalising possibility" and states that "there is little data on heuristic search techniques". Adaptive testing uses existing test data to generate further effective test data. It is a dynamic feed-back approach which involves the execution of the test software and monitoring the test effectiveness to decide the next action. It is this approach that has been developed in this thesis.

Chapter two defines some fundamental concepts and terms that are used throughout the thesis. A review of test data generators outlines each approach and indicates strengths and weaknesses. Greater attention is paid to adaptive test data generators. Chapter three describes the Heuristically Aided Testing System (HATS) and two heuristics it uses; the Direct Assignment (DA) heuristic and Alternating Variable (AV) heuristic. Chapter four discusses the application of HATS to a quadratic equation solving procedure, identifying problems with the two heuristics. Chapter five describes two further heuristics to overcome these problems; the Linear Predictor (LP) heuristic and Boundary Follower (BF) heuristic. The performance of these new heuristics is discussed. Chapter six discusses the application of HATS to a more complex procedure which performs triangle classification. Chapter seven focuses on the testing of loops and arrays through the use of a procedure that calculates the remainder after a division and a procedure that performs a linear search. Chapter eight presents the conclusions and further work.

2 Automatic Test Data Generation

2.1 Introduction

This chapter describes concepts that are used throughout this thesis, outlines testing methods and reviews automatic test data generators, concentrating mainly on adaptive generators.

2.2 Fundamental Concepts

Many of the concepts used in this section are from White and Cohen (1979). Variables in the test software are divided into three classes. If a variable accepts a value from the user or calling software, it is an *input variable.* If a variable provides a value to the user or calling software, it is an *output variable.* All other variables are termed *program variables.* When an element in the test software, i.e. a statement or a branch, is the focus of testing, it is described as being *considered.* The predicate associated with an element that is under consideration is termed the *considered predicate.*

Hecht (1977) describes a number of control flow models. The test software can be represented as a directed graph; the nodes represent a group of statements "such that no transfer occurs into a group except to the first statement in that group, and once the first statement is executed, all statements in the group are executed sequentially". The arcs represent transfers of control, or branches, between the nodes. Such a graph is termed a *control flow graph.* A restricted form of the control flow graph is the *control flow tree.*

White and Cohen (1979) highlight an important correspondence. "A program which has N input variables and produces M output variables computes a function which maps points in the N dimensional input space to points in the M dimensional output space". Focusing on the structure of the input space; "The input space is partitioned into a set of domains. Each domain corresponds to a particular executable path in the program and consists of the input data points which cause the path to be executed". More formally, they define an input space domain or simply *input domain* to be "... a set of input data points satisfying a *path condition*, consisting of a conjunction of predicates along the path". An *input data point* (a set of values; one for each input variable) may be referred to as a *point.*

Each input domain has a *boundary* which is "determined by the predicates in the path condition and consists of *border segments*, where each segment is the section of the boundary determined by a single simple predicate in the path condition".

If the test software contains loops, sequential selections or a combination of these, then a branch that is traversed after one of these constructs is executed has more than one partial path to the branch. A *partial path* in a control flow graph or tree begins at the entry or root node and ends at a non terminal node. For each partial path to a branch there is an *interpretation* that describes the branch's predicate as a function of the input variables. Each border segment determined by a predicate exists only in the input variables (dimensions) present in the predicate's interpretation. This number of dimensions may be from one to the number of input variables. Input variables in the interpretation of a partial path to a considered predicate are described as having *considered predicate influence.* However, input variables not in the considered predicate's interpretation but in the interpretations of earlier predicates in the partial path to the considered predicate are described as having *former predicate influence,* i.e. their values contribute toward arrival of control at the considered predicate. Input variables may be influential in both the former predicates and the considered predicate.

2.3 Testing Methods

A testing method typically comprises a test data selection strategy and a test data adequacy criterion. The *selection strategy* describes how test data shall be chosen to satisfy the *adequacy criterion* (or just *criterion*), which defines when testing may terminate. Weyuker (1986) states "Such a criterion represents minimal standards for testing a program, and as such measures how well the testing process has been performed". Consequently, when a criterion is satisfied the tester should have some degree of confidence that the software functions “acceptably”. Later in her paper, Weyuker discusses the relationship between an adequately tested program and a correct one. She states "An initial reaction might be that they should be intimately connected, perhaps even that an adequately tested program should be correct. But the purpose of testing is to uncover errors, not to certify correctness".

Many testing methods have been developed and are well described in Roper (1994). The methods typically fall into two classes, functional and structural, however some methods may fall into both classes. *Functional* (*black-box*) methods are driven by the test software's specification. *Structural* (*white-box*) methods are driven by the test software's code. Such methods are used in this thesis and are now highlighted. *Statement testing* and *branch testing* aim to select test data that cause execution of every source language statement or every decision outcome at least once. *Path testing* aims to cover every path in the test software. Even for small programs containing loops, path testing is unrealistic (Myers, 1979), so some subset of paths is chosen. A subset of paths can be selected that collectively satisfy some other criterion e.g. branch or statement testing. *Domain testing* (White and Cohen, 1980) aims to select points on and next to the border segments of the test software's input domains. *Mutation testing* (Budd, 1981) produces many similar versions of the original test software by introducing a single, syntactically correct change. The new programs are called *mutants* and it is mutation testing's aim to select test data that cause a difference in outputs between the original and mutant programs. When there is a difference, the mutant is described as being *revealed.* A set of mutants can be created so that the test data required to reveal them is equivalent to other testing methods.

A subsumption ordering of structural testing methods exists (Ntafos, 1988). This illustrates the relative difficulty of selecting test data to satisfy the criteria. Statement testing is the easiest to satisfy with branch testing next. Path testing is the hardest to satisfy. Other methods exist between path and branch testing.

2.4 Test Data Selection

Myers (1979) defines testing as "the process of executing a program with the intent of finding errors" and goes on to say "This definition ... implies that testing is a destructive process, even a sadistic process, which explains why most people find it difficult", and later reinforces that "...program testing is inherently an extremely difficult task". Since it is easy to measure how well a set of test data has satisfied an adequacy criterion, the difficulty lies in test data selection. This is confirmed by Roper (1994) who states "Many of the [testing] techniques are not prescriptive in their selection of test data". Hence a considerable degree of creativity is required, generally without specific guidance, for an essentially destructive process.

Myers (1979) argues "programmers cannot effectively test their own programs because they cannot bring themselves to form the necessary mental attitude : the attitude of wanting to expose errors". However, programmers typically conduct unit testing of their own code. Software testing is a repetitive task, which could be responsible for Graham (1991) stating "Many software developers seem to regard testing as boring". She indicates that a lot of tool support is required when software testing is regarded as deadly boring and states 90% of software developers have never been trained in testing methods.

Clearly, test data selection is hard and disliked. Further, it traditionally takes place late in the life cycle, when dead-lines may be stretched and delivery of the system being developed may be imminent. Clearly, automating test data selection should ease these problems.

2.5 Automatic Test Data Generators

Many different test data generators have been developed. This section classifies generators according to the approach used and identifies strengths and weaknesses. Emphasis has been placed on adaptive generators as the system described in this thesis falls into this class. A *static* generator does not execute the test software, however a *dynamic* generator does.

2.5.1 Random Test Data Generation

It is simple and cheap to randomly generate test data. Further it is stressing to the test software as unusual points may be used. However it is inefficient, if not virtually impossible, with small solution domains (Moranda, 1978). Nevertheless, it is a useful approach (Duran and Ntafos, 1984) and is a base line for comparison with other generators.

2.5.2 Syntax Based Test Data Generation

The input to the test software is formally described by a grammar. Test data are then generated from the grammar. This approach can produce large quantities of diverse functional test data. However, if the grammar has to be developed for testing this can be costly and time consuming. Burgess (1993) provides more detail and Ince (1987), a review.

2.5.3 Specification Based Test Data Generation

Test data are generated from a formal specification of the test software. Different systems have been developed (McMullin and Gannon, 1983; Denney, 1991; Furukawa, et al, 1985; Richardson and Clarke, 1981). The oracle problem (Weyuker, 1982) does not exist with specification based generators since the output from the test software can be compared against an executable version of the test software's formal specification. However, for a program of some size special skills and a considerable degree of effort is required to produce a formal specification.

2.5.4 Test Data Generation by Symbolic Execution and Solution of Produced Constraints

A path in the test software is specified then symbolically executed. Symbolic execution produces a set of constraints for the path, in terms of the input variables (an interpreted path condition). Linear or non-linear programming techniques attempt to solve the constraints. If successful, a point is produced for the path selected, otherwise the path is infeasible or the constraint-solver is not powerful enough. A number of path-based systems have been developed (Clarke, 1976; Howden, 1977; Ramamoorthy, et al, 1976; Voges, et al, 1980).

Offutt and Seaman (1990) describe an approach that symbolically executes the path to a mutated statement and produces further constraints that will reveal the mutation when test data that satisfy all the constraints are found.

Symbolic executors only require the program code to operate and can produce a minimal test data set to satisfy the specified criterion. However, there are several difficulties (Coward, 1988; Schmitz, et al, 1980). The number of times a loop iterates must be known and this is particularly difficult when this number of iterations is dependent upon input variables. Where subprograms are used in the test software further methods must be used to handle this. When an array is referenced by an input variable or is dependent upon at least one input variable, symbolic execution cannot proceed beyond this statement as the array element referenced cannot be identified uniquely. Many paths in software are infeasible (Hedley and Hennell, 1985), however the feasibility of a path will not be discovered until an attempt is made to solve the path condition. Since symbolic execution is a static approach focusing solely upon the test software, environmental faults may not be revealed that would be, perhaps coincidentally, with a dynamic approach.

Inamura (1989) has proposed a trial-and-error method as an alternative to linear and non-linear programming techniques. It analyses the path condition and uses constrained random test data generation and input variable backtracking to find a solution. A solution to array elements referenced by an input variable is proposed, but has limitations including being unable to operate upon conditions involving equality.

2.5.5 Adaptive Test Data Generation

Adaptive test data generators are feedback systems which use existing points to produce further points in an attempt satisfy some criterion. Searching and optimisation techniques are used to adapt the points.

Adaptive systems are typically dynamic; the test software is executed to produce feedback. Since the actual variable values are available, adaptive techniques do not suffer from the significant difficulties symbolic execution has. Software can be tested in its target environment, so that environmental faults may be revealed. Most adaptive systems know very little, if anything, about the function of the test software and assume very little about it. Consequently direct-search techniques (Gill and

Murray, 1974) for numerical optimisation, where only function values are compared, are suitable.

Adaptive test data generators can be split into two broad classes according to the adequacy criterion they aim to satisfy. These classes are fault-based and path-based.

A number of fault-based systems have been developed for use with Ballistic Missile Defence software. One of the earliest systems is due to Cooper (1976), who outlines the architecture of an adaptive testing tool and mentions the use of gradient, probabilistic and heuristic search techniques to bring about a maximal degradation in test software performance. This initial work is further developed by Andrews and Benson (1981) and Benson (1981). Both describe the use of Complex search (Box, 1965) for test data adaptation and executable assertions for performance evaluation. The objective is to maximise the number of violated assertions.

The path-based class is divided into two sub-classes. The first sub-class contains generators which have test paths selected before test data generation commences. The second, does not have paths preselected.

Miller and Spooner (1976) have developed one of the earliest systems, which preselects paths. Their system uses a variant of Rosenbroch's (1960) method to generate test data. The path consists of solely floating-point assignment statements interspersed with constraints that are derived from the predicates encountered on traversal of the path. More recently, Korel (1990a) has developed a system using numerical optimisation, dynamic data flow analysis and backtracking. The direct-search numerical optimisation technique, Alternating Variable (Glass and Cooper, 1965), modifies a single input variable at a time to minimise a function associated with a branch. Dynamic data flow analysis enables variables that are influential in the branch to be identified and ranked for modification. Backtracking focuses on branches where control has deviated from the selected path.

A system that converts test data generation into an unconstrained optimisation problem is presented by Gallagher and Narasimhan (1993). Instrumentation modifies the test software's conditions so that their outcomes can be controlled to force execution to take a specified path. Penalty functions (Adby and Dempster, 1974) are associated with each condition, so if control does not naturally take the required branch it has a large value corresponding to the distance it has "missed" the branch by. The objective function, which is the sum of the penalty functions, is optimised by a Quasi-Newton optimisation technique using the BFGS update (Minoux, 1986). This approach suffers, in that some of the test data generated will be forced down a non-natural path, so cannot be used to determine if a fault is present.

The main strength in this sub-class is in overcoming the inherent difficulties symbolic executors have. However, path selection and determining path infeasibility remains a difficulty. Complete path conditions can be quite complex, hence locating a solution to such a path may be difficult.

A number of systems fall into the second sub-class where paths are not selected before test data generation commences. Path or partial path selection is conducted during test data generation or is not explicitly considered. Prather and Myers (1987) acknowledge preselected path infeasibility and have observed that there is an intrinsic interplay between path selection and test data generation which can be exploited. With this foundation Prather and Myers have developed the Path Prefix testing strategy for branch coverage. The shortest partial path (path prefix) and corresponding point that traversed the partial path to an untraversed branch, is selected. The point is modified so that the untraversed branch is traversed (the decision is reversed). Here the closeness phenomenon is observed where test data that have traversed a branch are "close" to traversing the alternative branch. A method of inversion using back substitution to produce a decision reversing point, is mentioned, and gradient techniques are suggested as a potential solution. Branches are covered either by directly being considered or as a consequence of a decision being reversed (collateral coverage).

Kundu (1979) developed the earliest system, SETAR. It first executes an arbitrary point and stores the path traversed. This path is then symbolically executed to determine the path condition. A point is then found that violates at least one constraint in each of the previously traversed paths' conditions. This point is then executed and the cycle is closed. A method to overcome arrays referenced by an input variable is described. It is unclear when test data generation would stop as there will be a great number of violable constraints in test software containing loops.

Korel (1990b) describes a modification to his earlier system (Korel, 1990a) where preselection of paths has been eliminated and node coverage is attempted. The branches to a required node are classified; *critical* when branch traversal cannot lead to the required node, *required* when branch traversal leads closer to the required node, *semicritical* when branch traversal will need to iterate a loop once more to potentially take the alternative required branch and *nonessential* when branch traversal does not affect control leading to the required node. Execution is monitored so that if control deviates from the exact partial path to the required node then execution is terminated when the deviation occurs. The Alternating Variable search attempts to find a point that traverses the alternative branch at the deviation. Consequently, bringing control closer to the required node. Preservation of the exact partial path to the required node is later relaxed. If traversal of a nonessential branch is taken, execution continues. However, on traversal of a critical or semicritical branch, execution is terminated.

Deason, et al (1991) present a system that uses test data generation rules gleaned from software testing experts (DeMillo, et al, 1978; Howden, 1987) to satisfy structural condition-based criteria. The ten rules generate test data randomly or through analysis of the condition under test and / or through the use of the two closest points to the condition's boundary. A related system is described in Cross, et al (1991). Here the Path Prefix testing strategy (Prather and Myers, 1987) has been utilised to achieve branch coverage. The ten rules of the previous system have been replaced by four heuristics. The first two heuristics use symbolic simplification to derive an input variable's value at a condition's boundary with all remaining input variables held constant. Only values for influential input variables are generated, although how a condition's influential input variables are determined is not discussed. If the best point cannot be improved upon and the branch remains untraversed then the third and fourth heuristics modify the non influential input variables by 10% and reapply the first two heuristics.

The systems in this subclass incorporate some element or are a variation of the Path Prefix testing strategy, hence have similar strengths and weaknesses. Path infeasibility still presents a difficulty, but not to the extent of systems with paths preselected. It is known, at least, that the partial path to a condition is feasible. However, relative infeasibility, may occur where an absolutely feasible branch is untraversable due to the constraint of satisfying a partial path to the branch.

The system described in this thesis falls into this sub-class. The motivation for research in this area was provided by Ince (1987) who states "A number of questions about adaptive testing remain unresolved. There is still little data about its effectiveness. No work has been reported on using other measures of test effectiveness such as branch coverage, segment coverage or statement coverage.". He continues, "In particular there is little data on heuristic search techniques" and concludes, by describing adaptive testing as a "tantalising possibility".

Prather and Myers (1987) mention an inversion process that satisfies a considered condition by inverting values for all program variables back through the program logic to produce a point that will cause the condition to be satisfied. Prather and Myers, describe the inverse problem as "inherently difficult" and little work on the approach has been published since. However, they identify the use of techniques that can exploit the closeness of one partial path to another and discuss the use of a gradient technique.

3 The Heuristically Aided Testing System

3.1 Introduction

The Heuristically Aided Testing System (HATS) uses a library of adaptive heuristics in an attempt to satisfy a test criterion. A *heuristic* contains a rule or a series of rules that describe how test data should be adapted. The rules that a heuristic embodies, serves to distinguish it from other heuristics. A library of heuristics is proposed, so that the diverse functions and constraints present in software can be handled. HATS aims to achieve branch coverage of the software tested. Being a heuristic approach however, no coverage guarantee can be given.

HATS is a dynamic approach. The test software is executed and untraversed branches are selected for consideration during testing. Executing the test software enables the actual output to be obtained and may reveal failures. Dynamic branch selection overcomes the static branch selection problems, where paths must be chosen from a possible infinity and many paths are infeasible (Woodward, et al, 1980). A further benefit is that untraversed branches may be covered coincidentally by test data generated with the aim of traversing some other branch.

A model of the test software stores the test data generated and used by HATS. This chapter first discusses some important concepts underlying HATS, then describes the complete system, focusing on the two most important components : the HATS harness and the heuristics.

3.2 Objectives of HATS

HATS has three objectives

- to generate test data which satisfy the chosen testing criterion
- to offer an environment to support a variety of experimental heuristics
- to promote understanding of the adaptive approach

3.3 The Closeness Phenomenon and its Exploitation

A decision in a program can be either true or false. Each of these outcomes causes control to traverse a branch in the program and the corresponding branch in the program's control flow model. Prather and Myers (1987) observe that a point which has traversed a branch is “close” to traversing the alternative branch. By altering this point in some fashion the alternative branch may be traversed. Prather and Myers (1987), quote Beizer (1983) and Deutsch (1982), who make a similar observations.

By exploiting this closeness phenomenon, it is unnecessary to derive information on a partial path's function to generate test data. The *partial path function* is created by the assignment statements encountered on the partial path to the branch being tested, described in terms of the input variables. HATS treats each partial path function it encounters as a black-box. Only the input variable values and a value for the considered branch's predicate (section 3.5.5) are used.

3.4 HATS Architecture

HATS consists of three main components; the Modeller, Instrumentor and HATS harness. Figure 3.1 shows HATS's components and data flow. The test software is provided to the Modeller and Instrumentor. The Modeller generates a control flow tree. The Instrumentor (section 3.5.3) adds statements to the test software which record the control flow, program values and failure discoveries in the control flow tree.

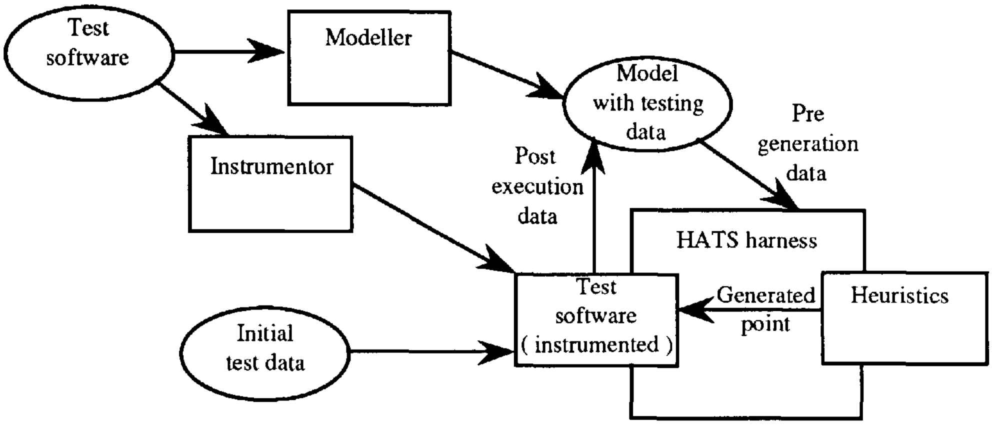


Figure 3.1 - HATS Architecture

First some initial points must be selected and executed by the test software. There are no restrictions on how to select the initial points or how many points to select. Initial points are typically selected by hand or randomly.

After execution of the initial points, the HATS harness operates. The HATS harness consists of the heuristics, support functions for the heuristics and the test software. The HATS harness first searches the control flow tree for an untraversed node. When one is found the harness selects a heuristic to consider this node.

The chosen heuristic generates a single point which is executed upon by the test software and updates the control flow tree. Then HATS harness takes control again and makes a number of decisions. If the considered node has been traversed then a further untraversed node is searched for. If the effort expended on a considered node exceeds a threshold, then it is deemed infeasible and a further untraversed node is searched for. Otherwise, the current heuristic is invoked again and its search continues.

The heuristic evaluates the data in the control flow tree toward generating a further point or terminating. If the heuristic generates a further point, the test software executes upon it. If the heuristic terminates the HATS harness chooses another heuristic to continue consideration of the node. If there are no more heuristics left then the node is deemed infeasible. This closes the cycle round the HATS harness, the heuristics and the test software.

3.5 Testing Criterion, Software Model and Instrumentation

3.5.1 Testing Criterion

Of the many testing criteria in existence, structural branch coverage (Roper, 1994) has been selected. This is the most commonly used structural criterion (Tai, 1990). The HATS harness attempts to generate test data that covers every branch left untraversed by the initial points, at least once.

3.5.2 Software Model

A software model (Hecht, 1977) is an abstraction of the test software and represents some aspect of the software, for example control flow or data flow. HATS uses a software model to store and retrieve data during the testing process. This data includes input values, output values, values of predicates, revealed failures and the flow of control taken through the test software. Each time the test software is executed this data is recorded in the model. The HATS harness uses the data in the model to select an untraversed branch for consideration by a heuristic and to gain feed-back on the progress of a heuristic. A heuristic uses data in the model to initiate testing of a branch and to gain feedback while a branch is being tested.

A control flow binary tree (Booch, 1987) is to be used to model the test software. Figure 3.2 shows an example procedure and its control flow tree.

The procedure TEST_ME has two integer input variables, A and B, and a string output variable, MSG. The procedure calculates A as a percentage of B and checks if the result is greater than or equal to 95%. If so, the message "OK" is returned, otherwise "NOT OK" or "B=0" is returned. The control flow tree clearly shows that there are three paths and four branches in the procedure.

Section 2.2 outlines the correspondence between the test software and its control flow model. For example the group of statements from line 10 to line 14 is represented by node 3. Note that the beginning of the "if" statement (line 14) is the group's terminator. A group of statements are represented more than once in the control flow tree when there are different partial paths to the group. A node may consist of zero statements so that branch testing can be satisfied.

```
Control flow tree node number
   Line number
        Ada statements

    1  procedure TEST_ME (
    2    A, B     : in INTEGER;
    3    MSG      : out STRING ) is
    4    C, D, E  : FLOAT;
    5    F        : INTEGER;
1 [ 6  begin
  [ 7    if ( B = 0 ) then
2 [ 8      MSG := "B=0  ";
    9    else
  [ 10     C := FLOAT ( ABS ( A ) );
  [ 11     D := FLOAT ( ABS ( B ) );
3 [ 12     E := C / D;
  [ 13     F := INTEGER ( E * 100.0 );
  [ 14     if ( F >= 95 ) then
4 [ 15       MSG := "OK    ";
    16     else
5 [ 17       MSG := "NOT OK";
    18     end if;
    19   end if;
    20 end TEST_ME;
```

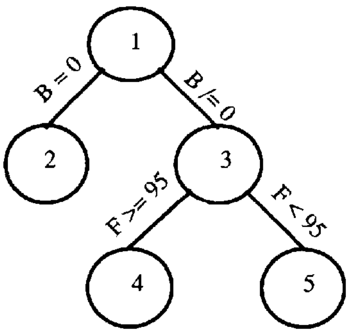


Figure 3.2 - The TEST_ME procedure and its control flow tree

Each node in the control flow tree, except the root node, holds data on the incoming branches predicate, e.g. data for predicate (B=0) is held in node 2 (figure 3.2). A predicate can be of the form E1 <rel> E2, where E1 and E2 are simple expressions and <rel> is a relational operator. HATS has classified predicates and presently recognises two types. First, where E1 is an input variable and E2 is a constant, and second, where the predicate is some other form to the first. The predicate type is stored in the corresponding control flow tree node and helps the HATS harness to select suitable heuristics.

Each node in the control flow tree has a linked list attached to it which holds data on a point that has caused traversal of the incoming branch. An element in the root node's linked list holds the input point, output point, path taken and a failure code. An element in a non-root node's linked list holds the input point and the node's predicate value (section 3.5.5). When a node is traversed, instrumentation (section 3.5.3) stores data in the corresponding node in the control flow tree. Figure 3.3 shows the linked lists in the TEST_ME procedure's control flow tree after two points have executed. For the sake of clarity only the input points are shown in each linked list's element. This clearly illustrates how the nodes are updated as control propagates through the test software.

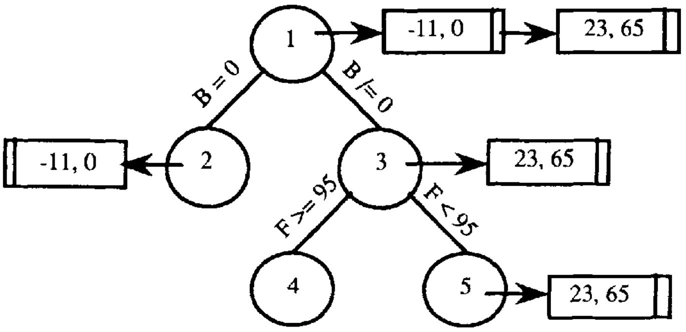


Figure 3.3 - TEST_ME's control flow tree showing each traversed nodes' linked list

3.5.3 Instrumentation

Instrumentation (Huang, 1978) is placed in strategic locations in the test software, to record essential data in the control flow tree. Four types of instrumentation are used, pre-execution, branch, postexecution and failure. Pre-execution instrumentation is placed so that it operates before the test software does (between lines 6 and 7, figure 3.4). It creates a new element in the root node's linked list, places the input point into the element and adds the root node identifier to a list of nodes traversed. Branch instrumentation is placed immediately after a branch in the test software (between lines 7 and 8, 9 and 10, etc., figure 3.4). As control propagates through the test software, branch instrumentation moves a pointer to the corresponding node in the control flow tree and creates a new element in the node's linked list which holds the input point and branch's predicate value (section 3.5.5). Further, the instrumentation adds the identifier of the node entered to a list of nodes traversed.

Postexecution instrumentation is placed to operate after the test software has completed (between lines 19 and 20, figure 3.4). It updates the root node's linked list element created by the pre-execution instrumentation with the output point and the path taken (list of nodes traversed). Failure instrumentation may be placed in various test software locations. These locations depend upon the failure trapping features provided by the language being used. This instrumentation traps a failure, preventing its effect upon the test software propagating to the HATS harness, and updates the corresponding element in the root node's linked list with a failure code.

```
1  procedure TEST_ME (
2   A, B     : in INTEGER;
3   MSG      : out STRING ) is
4   C, D, E  : FLOAT;
5   F        : INTEGER;
    PV2,PV3,PV4,PV5:INTEGER;PATH:PATH_TYPE;
6  begin
    INSTR_before_exec(A,B,PATH,1);
7   if ( B = 0 ) then
     PV2 := (B-0);INSTR_move_left(A,B,PV2,PATH,2);
8    MSG := "B=0  ";
9   else
     PV3 := (B-0);INSTR_move_right(A,B,PV3,PATH,3);
10   C := FLOAT ( ABS ( A ) );
11   D := FLOAT ( ABS ( B ) );
12   E := C / D;
13   F := INTEGER ( E * 100.0 );
14   if ( F >= 95 ) then
      PV4 := (F-95);INSTR_move_left(A,B,PV4,PATH,4);
15    MSG := "OK   ";
16   else
      PV5 := (F-95);INSTR_move_right(A,B,PV5,PATH,5);
17    MSG := "NOT OK";
18   end if;
19  end if;
    INSTR_after_exec(MSG,PATH);
20 end TEST_ME;
```

Figure 3.4 - Procedure TEST_ME with instrumentation

3.5.4 Selecting Branches to Test

The order that branches are selected for consideration is important. A top-down breadth-first search for untraversed nodes (which correspond to branches) in the control flow tree is to be used. This search reduces the risk of control deviating from a partial path to the considered node by selecting untraversed nodes that are closest to the first statement executed in the test software. Control flow along the partial path to the considered node is altered when an input point satisfies an alternative predicate and is termed an *upper-deviation.* By considering nodes higher in the control flow tree first, there is greater potential for *coincidental* traversal, where an untraversed node that is not the considered node, is traversed. This may take place whether or not an upper-deviation has occurred. Prather and Myers (1987) make similar observations they describe as collateral coverage.

3.5.5 The Input Point Closeness Metric

This metric indicates how close a point is to a boundary defined by the considered branch's predicate. The metric is termed a *predicate value* and is defined as (E1 - E2) which is derived from branch predicates of the form (E1 <rel> E2) (section 3.5.2). Figure 3.4 shows the predicate value expressions in the TEST_ME procedure.

The predicate value is relative to the condition it was derived from. Only predicate values for the same condition can be compared. To compare predicate values from different conditions would be meaningless. If two predicate values for the same condition are compared, the point whose predicate value is closest to zero, is the closest of the two points to the input domain boundary defined by the condition. Input domain boundaries exist close to or directly upon points that have a predicate value of zero. However, if the predicate value is used as a guide this may cause some difficulties. An *interfering predicate* will cause an upper-deviation. Consequently, no predicate value will exist for the considered node for the point used. Hence, only an *expected boundary* location can be predicted using the predicate value.

A heuristic can use this metric to exploit the closeness phenomenon. It allows a heuristic to determine which side of a boundary a point is on, from the polarity of the predicate value. This may be useful when considering an equality predicate, as it allows points either side of the boundary to be distinguished.

Authors of other test data generators have proposed similar metrics, which are based on closeness and enable the comparison of points. Deason, et al (1991) propose a metric which calculates the percentage difference between the left and right hand sub-expressions of a condition. Korel (1990a) transforms a branch's predicate to an equivalent predicate that is related to zero.

3.5.6 Adequacy Criterion Influence on the Software Model and Instrumentation

There is an important relationship between the adequacy criterion adopted, the software model and instrumentation used. The criterion defines the form of the software model. The software model must explicitly represent, in some way, each necessary sub-goal, which collectively satisfy the criterion. For example, with LCSAJ testing (Woodward, et al, 1980), each LCSAJ would need to be explicitly represented in the software model. Structural criteria are commonly used, however models for data flow and other non-structural criteria may be used (Furukawa and Ushijima, 1987; Infotech, 1979). Such models are statically generated. Storage for HATS's data must be incorporated into the model.

Traditionally, a criterion is expressed in terms of paths or subpaths in a control flow graph. Generally the test software's control flow graph, or some variation of it, would be used. The software model's form defines the instrumentation's function and location in the test software. Instrumentation must record essential data on the behaviour of the test software and necessary program values.

3.5.7 Testing at Various Levels of Abstraction

HATS is not restricted to unit testing. This section describes how higher level testing is achieved. The all branches criterion tests software at a low level. The software modelling approach adopted enables software to be tested at other levels of abstraction. For example, in integration testing, it is important to check interfaces and code coverage across many procedures. If this criterion can be specified and explicitly represented within some model of the test software then HATS can test it. Presently, there is only the input point closeness metric available, however other metrics could be devised. A suitable model for integration testing may be the static procedure call hierarchy.

It is not a prerequisite that models are derived from the test software's code. If we consider the design as a model, then this could be used, together with some criterion to test the software produced from the design.

3.5.8 Reuse of the Software Model and its Data

Software evolves over its life time. Consequently, the software's test data must also evolve. The software modelling approach embodies this evolution. After the application of HATS, the model stores data on the behaviour of the test software. This data may be rationalised, so that only essential input points and associated data are stored. When the software is changed unit testing of the new parts and regression testing of the system would normally be conducted. The model must be updated to reflect the changes made and HATS applied. A regression testing system could be constructed which selects appropriate input points from the model and analyses its old and new behaviour of the test software from the model. Data from the model may also be used in other processes.

3.6 The HATS Harness

The HATS harness controls the heuristics and test software, and provides support functions for the heuristics. Figure 3.5 shows each component of the harness and its iterative nature.

Search for an untraversed node uses a top-down breadth-first search (section 3.5.4) of the control flow tree for an untraversed node. *Select a heuristic* uses a selection hierarchy to choose a heuristic to consider the untraversed node found. The *heuristic selection hierarchy* places an order on the application of available heuristics. At the top of the hierarchy are general heuristics and specific heuristics for immediately identifiable problems. Lower in the hierarchy are increasingly problem-specific heuristics. Select a heuristic starts at the top of the hierarchy and, as heuristics fail, chooses heuristics lower down. *Apply a heuristic*, initialises the heuristic and iterates it a number of times on the test software. An *iteration* involves one complete execution of the test software.

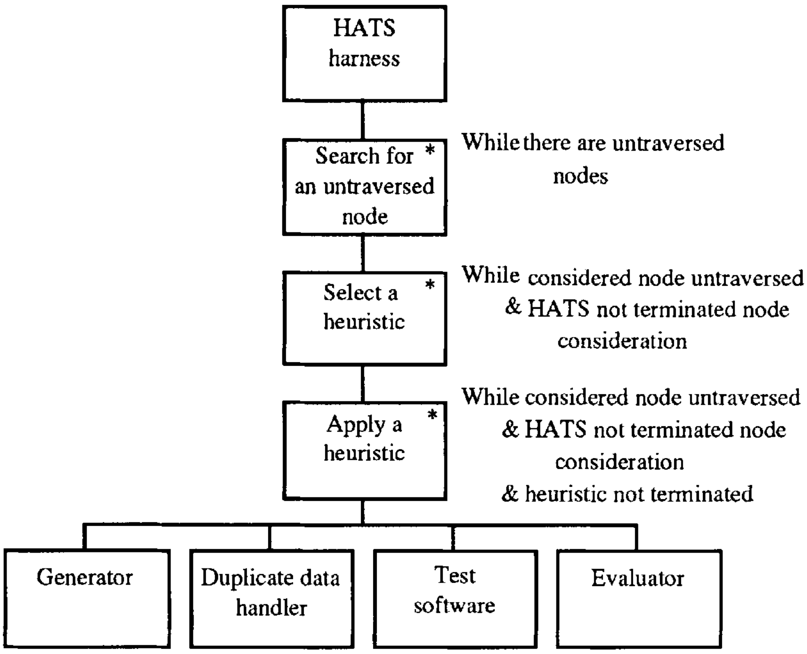

HATS harness
Search for an untraversed node *
While there are untraversed nodes
Select a heuristic *
While considered node untraversed & HATS not terminated node consideration
Apply a heuristic *
While considered node untraversed & HATS not terminated node consideration & heuristic not terminated
Generator
Duplicate data handler
Test software
Evaluator

Figure 3.5 - Structure diagram of the HATS harness

The *generator* exercises the heuristic's instructions to generate a point, termed the *execute point*, from another point, termed the *generate point*. The generated execute point may have already been executed upon the test software and be stored in the test software model. If required, the *duplicate data handler* will produce an execute point that has not been executed before in the current run. Not all heuristics require such an execute point. The *test software* has instrumentation included and executes upon the execute point.

The *evaluator* does three tasks. First it determines a traversal status from the path taken by the execute point. *Traversal-status* can either be upper-deviation, sibling-traversal or node-traversal. *Sibling-traversal* indicates that the execute point has traversed the sibling node to the considered node. Sibling-traversal can be further refined into *positive-sibling-traversal*, where the predicate value is positive, and *negative-sibling-traversal*, where the predicate value is negative. *Node-traversal* indicates that the considered node has been traversed.

The second evaluator task enables a heuristic to review its progress. The heuristic either produces instructions for the generation of the next execute point or elects to terminate. The final task is node termination evaluation. If a node has been considered for too long the node is deemed unreachable. The unreachable nodes component is called which adds the considered node and its descendants to a store. Search for an untraversed node will not consider any nodes in this store. The *node iteration threshold*, specifies the maximum number of iterations the HATS harness can make on any considered node.

3.7 Heuristics

This section discusses some important issues for heuristics and highlights their structure. The heuristics described in this thesis are *deterministic*, in that their behaviour will always be the same given the same point to start with and test software set up. There is no random influence once the heuristic commences operation. In addition the heuristics are *direct search* techniques (Murray, 1972), in that they are based on the comparison of predicate values. They require no prior knowledge about the partial path function they are operating with.

All the heuristics described in this thesis depend upon the test data available when they commence. Since the heuristics are deterministic it would appear to follow that the greater the number of points available the higher the chance of finding a solution.

A heuristic may alter any of the input variables. The heuristics described in this thesis modify only one or two input variables at a time. The test software may have many input variables, hence some method of selecting the input variable(s) to modify and dynamically referencing them is necessary. The method adopted allocates each scalar (atomic) input variable with a run-time identifier. The allocation of run-time identifiers is based on the order that input variables appear in the test software's parameter list. Correspondingly, input variables are considered for modification in the same order; run-time identifier order.

A significant responsibility of a heuristic is to terminate (Gill and Murray, 1979) as soon as a heuristic recognises that it is unlikely to locate a solution. Determining when, is potentially a complex task. There is the chance that after terminating the heuristic would have located a solution given at least one more iteration. However, if little progress has been made then the risk of this occurring is minimal. Nevertheless, if a heuristic has been operating past its best termination point then there is the possibility that it has generated test data that another heuristic may use to success and coincidental coverage may have increased.

For a heuristic to be used in the HATS harness it must provide up to four components. These components are now described and the flow through a heuristic illustrated (figure 3.6).

The*first iteration set-up* component initialises the heuristic when the HATS harness chooses to use it. It sets all the heuristic's variables, selects the first generate point and specifies generation instructions for the heuristic's first iteration.

The *generator* component applies the instructions given by the first iteration set-up component or evaluator component to produce the execute point from the generate point.

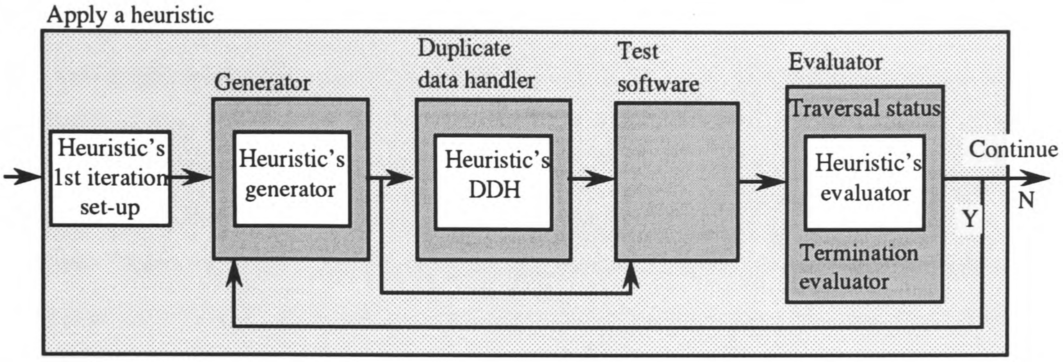


Figure 3.6 - Components and flow in a heuristic

The *duplicate data handler* is an optional component and is only provided and used if the heuristic requires a unique execute point to be produced every iteration. The handler modifies the execute point in an attempt to produce a unique execute point that does not upset the heuristics approach.

The *evaluator* component reviews a heuristic's progress and either produces new generation instructions and selects a generate point, or terminates the heuristic. The actions a heuristic's evaluator takes are generally based on the traversal status of the execute point.

A time-consuming problem in software testing is determining if the output for an input point is correct with respect to the test software's specification. This is known as the oracle problem (Weyuker, 1982). It has been overcome in the HATS harness by incorporating postconditions in the test software. If the postconditions are violated then a failure is recorded in the software model.

3.8 Selection of Initial Points

Initial points can be selected in any way and in any quantity. There are no restrictions except that it must be executed upon the test software so that essential data is recorded in the software model. Normally however, only a small number of points will be selected, typically one. This places the emphasis of generating adequate test data upon HATS.

3.9 The Direct Assignment Heuristic

Test data generation for predicates that include input variables and constants is an obvious and simple task for a heuristic. However for a human, it may not be so obvious. The Direct Assignment (DA) heuristic exploits input variables that appear in such predicates.

3.9.1 Overview of the Direct Assignment Heuristic

For a predicate to qualify, the input variable appearing in the predicate must not be redefined on any path from the first statement executed in the test software to the predicate. The expressions in the predicate must be of a specific form. Using the predicate form defined in section 3.5.2, E1 must be an input variable and E2 a constant. Static analysis of the test software determines if a predicate qualifies. If so, the predicate's relational operator and constant value are stored in the control flow tree. When an untraversed node controlled by such a predicate is considered the DA determines a value for the input variable using the data stored in the control flow tree, that should cause traversal of the considered node. A point, with the determined value for the input variable appearing in the predicate, is executed and the result evaluated. If the considered node is not traversed then it and its descendants are deemed infeasible.

An example of the DA's application using the TEST_ME procedure follows. The single initial point (7, 20), where the first value defines input variable A (7) and the second B (20), traverses the path 1, 3, 5 in the control flow tree (figure 3.2). Node 2 is the first untraversed node to be found by the top-down breadth-first search. The HATS harness selects the DA since it is suitable for node 2's predicate. The DA determines that a value of 0 for input variable B will cause traversal of node 2. A value of 7 for input variable A is gained from the first and only point to traverse node 2's sibling, node 3. TEST_ME is executed with the point (7, 0) and the path 1, 2 is traversed, satisfying the DA's goal.

The DA bears similarity to Howden's (1987) functional testing rules for conditional branching which have been automated by Deason, et al (1991).

3.9.2 Components and Functions of the Direct Assignment Heuristic

The DA's pseudo code can be found in appendix A3.1.

3.9.2.1 First Iteration Set-up, Generator and Duplicate Data Handler

Most of the DA's functionality operates before the first iteration since the heuristic takes only one iteration. The considered predicate's constant is assigned to the input variable appearing in the considered predicate. The input point generation instructions for the input variable involved are defined according to the considered predicate's relational operator, as shown in table 3.1.

Relational operator	Instructions
<	increase by 1
>, /=	decrease by 1
=, <=, >=	no change

Table 3.1 - DA's generation instructions for the considered predicate's input variable, defined by the considered predicate's relational operator

The remaining input variable values for the generate point are provided from the first point to traverse the considered node's sibling. The generator applies the input point generation instructions to the generate point to produce the execute point, which the test software will execute upon if it is unique. If the execute point exists in the control flow tree, the duplicate data handler records this and the HATS harness terminates.

3.9.2.2 Evaluator

If the DA causes traversal of the considered node then control returns to the HATS harness. Otherwise, if the node remains untraversed, the node and its descendants are deemed infeasible and added to a list of unreachable nodes. This takes place when an interfering predicate is encountered before the considered predicate, causing an upper-deviation.

3.10 The Alternating Variable Heuristic

When the considered predicate is some function of the input variables the Alternating Variable (AV) heuristic is applicable. The AV modifies an input variable in ever increasing steps in the most promising direction in an attempt to traverse the considered node.

3.10.1 Overview and Phases of the Alternating Variable Heuristic

Figure 3.7 is a partial input plane for an arbitrary program which illustrates the AV operating in an ideal situation. A *partial input plane* is a constrained two dimensional view of a program's input space. The program's input space may be any number of dimensions. If there are more than two input variables then the input variables not included in the plane must be held constant and their values shown. The arbitrary program has two input variables, X and Y, and a condition dependent upon both these variables. The condition defines the boundary which partitions the input space into two domains. Input points on the right side of the boundary cause considered node traversal. Whereas input points on the left side of the boundary and directly upon it, cause sibling-traversal. The AV's objective is to locate a point that causes considered node traversal (*solution point*) from the closest point causing sibling-traversal. The *closest point* is a previously executed input point which caused sibling-traversal and has a predicate value that is closer to the considered node's expected boundary than any other input point previously executed.

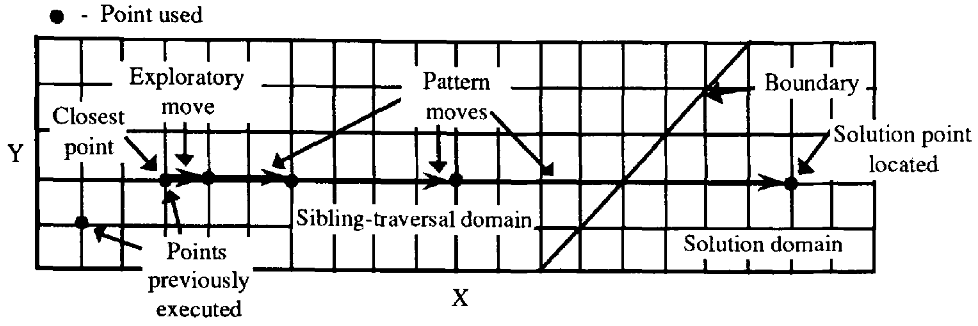


Figure 3.7 - Partial input plane showing the AV operating

The AV first has an *exploratory* phase where the direction to modify the considered input variable is discovered. Comparing the value of variable X in the two executed points it is clear to see that increasing X is the most promising modification direction. An exploratory move increases X and the resultant point is closer to the boundary and solution domain.

Now that a modification direction has been established the exploratory phase prepares for the pattern phase. The *pattern* phase modifies the considered input variable in the direction established with ever increasing steps. Figure 3.7 shows that the first pattern move is made from the exploratory point and is twice the size of an exploratory move (2). The second pattern move is twice the size of the previous (4) and finally the third pattern move is twice the size of the second (8). The final move crosses the boundary to a solution point. When no improvement is made in the pattern phase the modification direction and step size may be adjusted or the exploratory phase may take over again. If it becomes evident during either of the phases that no progress is being made modifying an input variable then the exploratory phase takes over on the next input variable.

The AV is a single-dimension version of Glass and Cooper's (1965) Sequential Search where the alternate routine has been omitted. The AV is a variation of Korel's (1990a) search procedure which is based on Sequential Search.

3.10.2 Components and Function of the Alternating Variable

The AV's pseudo code can be found in appendix A3.2

3.10.2.1 First Iteration Set-up, Generator and Duplicate Data Handler

The first iteration set-up component specifies that the current phase is exploratory, that the first input variable shall be considered and establishes a modification direction to use. If the considered node has only one sibling-traversal point, a default modification direction of increase by one is used. However, when there are two or more sibling-traversal points, a modification direction can be determined. The considered input variable values for the two sibling-traversal points with the closest predicate values to the expected boundary are compared. A further check is made to see if the expected location of the solution point lies between the two closest points. Table 3.2 shows how the exploratory modification direction is chosen. The generate point is defined as the closest point to the expected boundary.

Relationship of input variable values	Solution point between closest points	Instructions given
Closest < next closest	No	decrease by 1
"	Yes	increase by 1
Equal		increase by 1
Closest > next closest	No	increase by 1
"	Yes	decrease by 1

Table 3.2 - Determining exploratory phase input point generation instructions

The generator component produces the execute point by applying the input point generation instructions to the generate point. The execute point must be unique. If it is not, the duplicate data handler is used and its action is based on the modification direction. When increase, the considered input variable of the execute point is further increased by 1. When decrease, the considered input variable is further decreased by 1. This continues until a unique execute point is produced which is executed upon by the test software.

3.10.2.2 Evaluator

The evaluator component first checks if the execute point has traversed the considered node. If so, control returns to the HATS harness, otherwise evaluation is performed dependent upon the traversal-effect of the execute point and, when a sibling-traversal took place, the closeness of the execute point to the expected boundary compared to the closest point stored.

3.10.2.3 Sibling-traversal and Closer to the Expected Boundary

Entry to this evaluator indicates that the execute point has made progress toward the expected boundary location and possible locality of a solution point. The following example illustrates this evaluator and continues from the previous example in section 3.9.1. The initial point (7, 20) traversed path 1, 3, 5 and the DA generated a point that traversed node 2, which leaves only node 4 (F >= 95) untraversed in the TEST_ME procedure. The HATS harness selects the AV to consider node 4. The exploratory phase is used first. Since there is only a single point to cause sibling-traversal (7, 20), this is stored as the closest point. The default direction of increase is used and, as it is the exploratory phase, the step size is one. The first input variable A is considered for modification.

Figure 3.8 illustrates the moves made by the AV in a partial input plane. When a point is used by a heuristic, the point's considered predicate value is shown close to it.

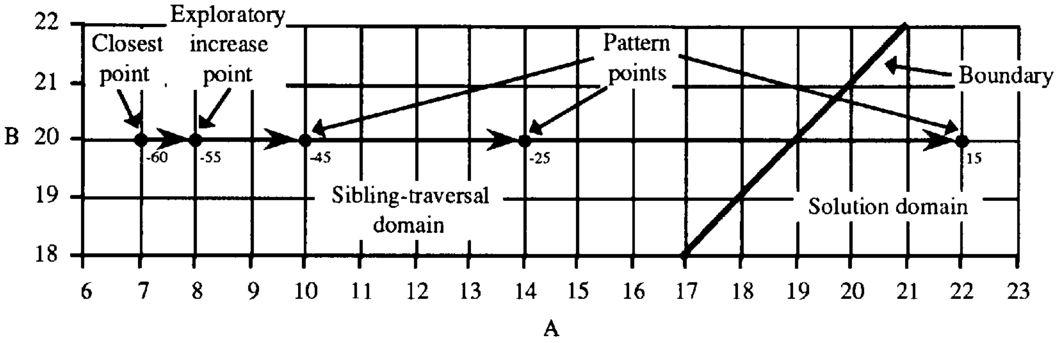


Figure 3.8 - Partial input plane showing moves made by the AV considering node 4 of the TEST_ME procedure

The exploratory move from the closest point (7, 20), produces a point that is closer to the boundary and solution points, hence the exploratory phase prepares for the pattern phase. For the next three iterations, ever increasing steps are taken. The first two produce sibling-traversal points that are closer to the boundary and the third locates a solution point (22, 20) for node 4.

The actions taken by this evaluator are now summarised. If the AV is in the exploratory phase then the pattern phase is prepared for. This involves doubling the present step size of one to two. The modification direction is not changed. If the AV is in the pattern phase then a check is made to see if the expected location of the solution point is between the executed point and the former closest point stored. If so, the modification direction is reversed and the pattern step is considerably decreased, otherwise the pattern step is doubled.

3.10.2.4 Sibling-traversal and Further from the Expected Boundary

Entry to this evaluator indicates that the execute point made no progress toward the expected boundary and possible locality of a solution point, hence suitable corrective action is taken. To prevent the AV from concentrating on one input variable when no progress can be made modifying it, the number of sibling-traversals further from the expected boundary for the considered variable are counted. If this exceeds a threshold then the considered variable is abandoned.

When the AV is in the exploratory phase a check is made to see if this is the first sibling-traversal further from the expected boundary for the considered variable. If so, the modification direction is reversed and the generate point is stored as the closest point. This check enables an exploratory move in the opposite direction. Searching in both directions may be expected if there was only one sibling-traversal point. There has to be at least two sibling-traversal points for a modification direction to be determined. If, in the exploratory phase, there are two sibling-traversals further from the expected boundary then the considered input variable is abandoned. Searching in both directions has not found a point closer than the closest point stored.

When the AV is in the pattern phase the last move has possibly overstepped a solution point. In an attempt to locate the solution, the modification direction is unchanged, the pattern step is considerably reduced and the generate point is stored as the closest point. Since it appears that a solution exists, it is anticipated that a solution will be found after a few iterations modifying the considered variable. However, should a solution not be found, then to prevent the AV from continuing to overstep the expected location of a solution, the number of further sibling-traversals produced by modifying the considered variable, whilst in the pattern phase, are counted. If this exceeds a threshold then the considered input variable is abandoned.

3.10.2.5 Upper-deviation

Entry to this evaluator indicates that a move by the AV, in either the exploratory or pattern phases, has overstepped an interfering predicate's boundary. This evaluator sets up the AV to "home in" on the interfering predicate's boundary where closer points and possibly a solution will be located.

When the AV is in the exploratory phase a check is made to see if this is the first upper-deviation on the considered variable. If so, the modification direction is reversed and the generate point is stored as the closest point. This enables an exploratory move in the opposite direction. If there are two upper-deviations in the exploratory phase, the considered input variable is abandoned. When the AV is in

the pattern phase, the exploratory phase is prepared for, which will "home in" on the interfering predicate's boundary. The step size is reduced to one and the generate point is stored as the closest point.
To prevent the AV from concentrating on one input variable when no progress can be made, the number of upper-deviations produced for the considered variable are counted. If this exceeds a threshold then the considered variable is abandoned.

3.10.2.6 Abandoning Consideration of an Input Variable

The new considered input variable is the next in line to the old considered input variable. If the old considered input variable is the last then the new is the first in the list. The modification direction and generate point are determined in the same fashion as the first iteration set-up (section 3.10.2.1).

3.11 Scope of Testable Software

To test the test software, its object code must be linked with the HATS harness's object code. Provided, this is possible and the test software has been modelled and instrumented correctly then a subset of most third generation languages can be used. However, since Ada has been used to develop HATS, if the test software is written in Ada this will reduce the risk of integration problems.

3.11.1 Testable Subset of Ada

This subset represents a starting point that will enable experience using HATS to be gained. There are several parts outlined, these being statements, data types, conditions, subprograms, input / output and subpath / path expressions.

- Following Barnes' (1989) statement classification, testable simple sequential statements are null, assignment and procedure call, and the testable compound sequential statements are if and case.
- Any data type may exist in the test software. However, data types of conditions and input / output is limited.
- A condition may only be of the simple form outlined in section 3.5.2, where each subexpression is of integer data type.
- Function or procedure subprograms may be tested. They may be compounded.
- Input to a subprogram must come through its parameters and be integer. Output may be of any type and is not necessary for it to be passed through the subprograms parameters.

- Subpath / path expressions are not restricted at all. They may be linear, nonlinear or a combination. They may be of any complexity.

Although loop statements have been omitted from this list they are incorporated in chapter 7.

3.11.2 Transformations

To model and instrument the test software, it is necessary to transform some of the instances of the statements mentioned above into a similar but logically equivalent form. Transforming a program, or adding instrumentation, has the risk of altering its original behaviour. This would naturally make the testing process worthless.
With "if" statements, all variations must be represented in the "if then else" form. Case statements must be transformed into a series of "if then else" statements. "If" statement conditions with logical connections, i.e. AND, OR, can be transformed to a number of "if" statements with simple conditions.

3.12 Automated and Manual Aspects of HATS

Only the HATS harness has been automated. Generation of the software model, instrumentation and generation of the initial test data are performed manually. However, their automation is well understood (Infotech, 1979).

4 The Quadratic Equation Solver Problem

4.1 Introduction

This chapter describes the application of HATS using the Direct Assignment (DA) and Alternating Variable (AV) heuristics to a quadratic equation solving procedure. Firstly, HATS generates test data for branch testing of the QUADRATIC procedure, secondly, HATS generates test data for Mutation Analysis (MA) of the procedure. Both branch testing and MA reveal the difficulty HATS has in generating a point that satisfies the only nonlinear equality predicate in the procedure, which has few points that do satisfy it. MA is harder to satisfy than the branch testing criterion and it extensively tests both the QUADRATIC and heuristics.

Control flow tree node number
Line number
Ada statements

```
   1  procedure QUADRATIC ( A, B, C         : in INTEGER;
   2                        X1, X2          : out COMPLEX;
   3                        QUAD_KIND       : out QUAD_TYPE ) is
   4    D:INTEGER; REAL_PART:FLOAT; IMAG_PART: FLOAT;
   5  begin
┌  6    SET (X1,0.0,0.0);
1  7    SET (X2,0.0,0.0);
└  8    if (A=0) then
2[ 9      QUAD_KIND := NOT_A_QUADRATIC;
  10    else
3┌ 11     D:=(B*B)-(4*A*C);
 └ 12     if (D>0) then
┌ 13        QUAD_KIND:=ROOTS_ARE_REAL_AND_UNEQUAL;
│ 14        REAL_PART:=(FLOAT(-B)+SQRT(FLOAT(D)))/FLOAT(2*A);
4 15        SET (X1,REAL_PART,0.0);
│ 16        REAL_PART:=(FLOAT(-B)-SQRT(FLOAT(D)))/FLOAT(2*A);
└ 17        SET (X2,REAL_PART,0.0);
  18      else
5[ 19       if (D=0) then
┌ 20          QUAD_KIND:=ROOTS_ARE_REAL_AND_EQUAL;
│ 21          REAL_PART:=FLOAT(-B)/FLOAT(2*A);
6 22          SET (X1,REAL_PART,0.0);
└ 23          SET (X2,REAL_PART,0.0);
  24        else
┌ 25          QUAD_KIND:=ROOTS_ARE_COMPLEX;
│ 26          REAL_PART:=FLOAT(-B)/FLOAT(2*A);
7 27          IMAG_PART:=SQRT(FLOAT((4*A*C)-(B*B)))/FLOAT(2*A);
│ 28          SET (X1,REAL_PART,IMAG_PART);
│ 29          IMAG_PART:=-IMAG_PART;
└ 30          SET (X2,REAL_PART,IMAG_PART);
  31        end if;
  32      end if;
  33    end if;
  34  end QUADRATIC;
```

Figure 4.1 - The Ada QUADRATIC procedure

4.2 The Integer Quadratic Equation Solver Procedure

The QUADRATIC procedure (figure 4.1) has three integer input variables A, B and C, representing the coefficients of the quadratic equation. There are three output variables; X1 and X2, of COMPLEX type, are the two roots of the given quadratic equation; QUAD_KIND, of QUAD_TYPE, is an INTEGER subtype and is constrained to the range 1 to 4. Each value in this range corresponds to one of four QUAD_TYPE constants; NOT_A_QUADRATIC, ROOTS_ARE_REAL_AND_UNEQUAL, ROOTS_ARE_REAL_AND_EQUAL or ROOTS_ARE_COMPLEX.

The procedure has three conditions. The first (A = 0), involves only an input variable and a constant. The other two conditions, (D > 0) and (D = 0), involve the result of a non-linear function of the input variables ($D = B^2 - 4AC$) and a constant (0). Program variable D, is local to the procedure and is integer.

Figure 4.2 shows that there are four distinct paths through the procedure and the longest path consists of four nodes. There are no iterations and the procedure has three branch nodes (1, 3 and 5).

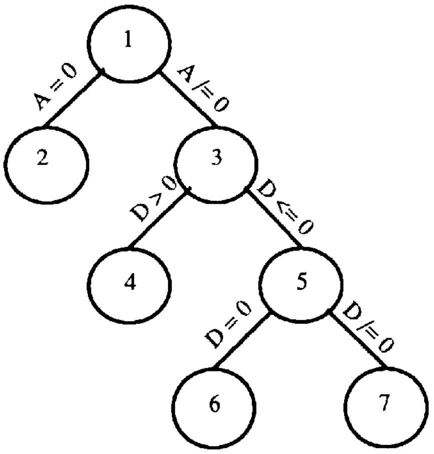


Figure 4.2 - Control flow tree of the QUADRATIC procedure

4.2.1 The QUADRATIC's Input Space

Table 4.1 illustrates the path conditions and the corresponding domain details.

Path	Path condition	% of input space domain occupies (range ±1000 for A, B & C)	Domain population
1,2	A=0	0.05%	4,004,001
1,3,4	A/=0 & D>0	62.6722%	5,021,301,640
1,3,5,6	A/=0 & D=0	0.0002%	15,384
1,3,5,7	A/=0 & D<0	37.2776%	2,986,684,976
		Total	8,012,006,001

Table 4.1 - The QUADRATIC's input domains

Paths 1, 3, 4 and 1, 3, 5, 7 have the largest domain population. Consequently, input points have a higher probability of traversing these two paths than paths 1, 2 and 1, 3, 5, 6. Path 1, 2, has a considerably smaller domain than paths 1, 3, 4 and 1, 3, 5, 7. Path 1, 3, 5, 6 has an even smaller domain than path 1, 2, which is sparsely located between the two largest domains. The form of these four domains is shown in figure 4.3.

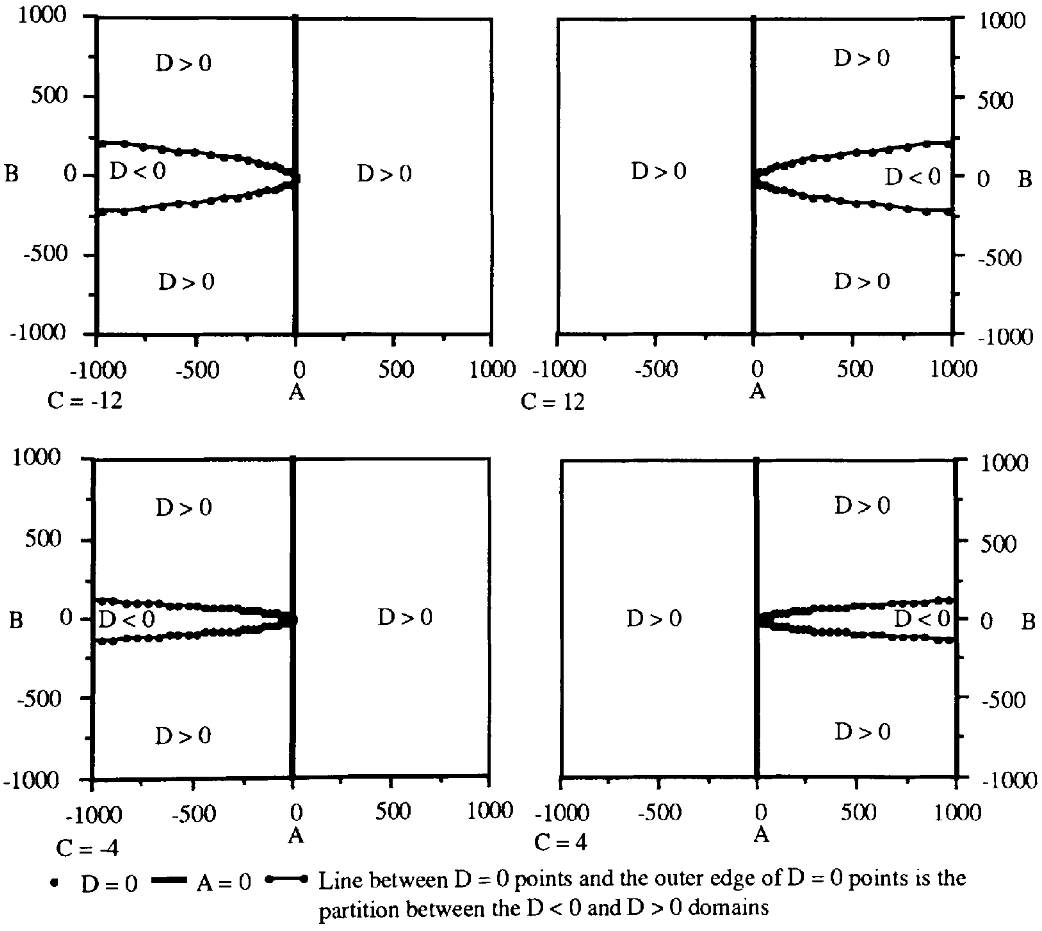


Figure 4.3 - Partial planes of the QUADRATIC's input space

If the data type of the QUADRATIC's variables were real then the (D=0) predicate's border would be pseudo-continuous, where input points approximating to the border, are contiguous. Such a procedure's input space would have three borders.. A border where points make A=0; a border lying on, but not including, points where D=0 (for the D>0 condition); and finally a border lying on, and only including, points where D=0 (for the condition D=0).

However, the QUADRATIC's variables are integer, and the (D=0) border consists of disjoint points that are rarely adjacent. A *notional border / notional boundary* lies in the place a border / boundary would be had real variables been used instead of integer. Therefore the integer QUADRATIC's borders are different to the same procedure with real variables. There is a border where points make A=0; a notional border lying through, but not including, points where D=0 (for the condition D>0); and finally a notional border lying through, and only including, points where D = 0 (for the condition D=0).

4.3 Branch Testing of the QUADRATIC

4.3.1 HATS Experimental Set-up

A single, hand selected point is used to start each HATS harness run (table 4.2). The values chosen for each of the three input variables are in the arbitrarily chosen range of ±1000.

HATS	Input variable		
run	A	B	C
Q1	-200	50	9
Q2	17	27	46
Q3	257	46	-63
Q4	699	103	675

Table 4.2 - Initial points for branch testing of the QUADRATIC

The node iteration threshold is 50 and the input variables are considered in the order A, B then C by the AV.

4.3.2 Run Q1

Six of the seven nodes were traversed. Table 4.3 shows the traversal results for this run. A traversal results table contains important information on a HATS run and can consist of up to four sections. The first section shows the path taken by the initial point. The second section, nodes considered and traversed, shows for each considered node traversed by a heuristic, the heuristic used, the number of iterations taken by the heuristic and nodes coincidentally traversed through the heuristic considering the node. The third section, nodes considered and untraversed, shows the same information as the previous section but for considered nodes that could not be traversed. The fourth section, nodes unconsidered (not shown in table 4.3), lists nodes that the HATS harness did not consider.

Initial point path		1,3,4	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	1	
5	AV	9	7
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
6	AV	50	

Table 4.3 - Run Q1 traversal results

The DA took a single iteration on node 2. On node 5, the AV took 9 iterations and generated a point that coincidentally traversed node 7. No solution could be found in the maximum 50 iterations using the AV on node 6. Table 4.4 gives a more detailed breakdown of what occurred whilst a node was being considered by a heuristic.

Considered node	Heuristic used	Total iterations	Number of iterations :				
			Variable modified			Upper - deviation	Duplicate data
			A	B	C		
2	DA	1	1	0	0	0	0
5	AV	9	9	0	0	0	0
6	AV	50	40	6	4	15	10

Table 4.4 - Run Q1's iteration summary for the nodes considered

Nodes 2 and 5 were traversed by a heuristic modifying only input variable A. On node 6 all three input variables are modified as the AV changed from modifying one variable to another as it was not making progress. Of the three input variables, A was modified the most. Only on node 6 were upper-deviations produced and duplicate data generated as a domain boundary was encountered.

The operation of the heuristics on each node is now described. On node 2, the DA produced the execute point (0, 50, 9) (where A=0, B=50 and C=9) from the considered predicate (A=0) and the initial point. On node 5, the AV had only the initial point (which traversed node 4) to generate from. Figure 4.4 shows how the AV modified only variable A to locate a solution to node 5 at (311, 50, 9). The figure represents an approximation of a partial input plane and shows the large domains of nodes 4 and 5. On the penultimate iteration, a point close to the notional boundary was generated. Since the point is in a domain that causes node 4 to be traversed, the AV continued, taking another step twice the size of the previous, to locate a solution point.

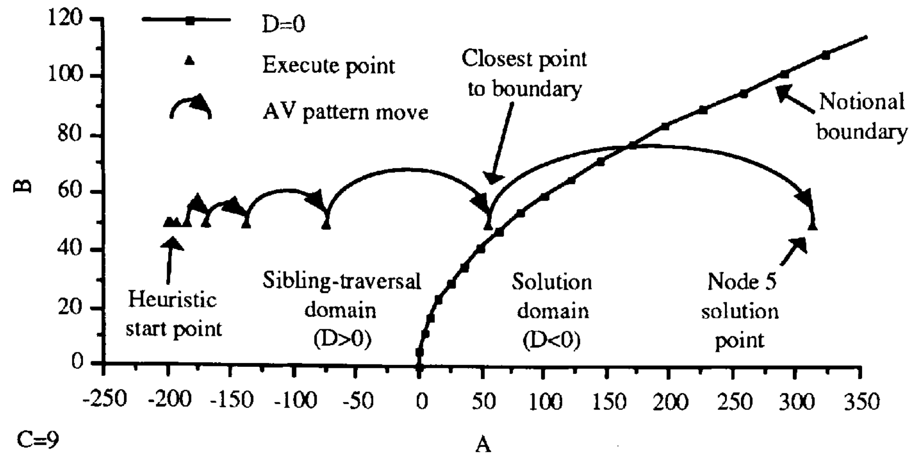


Figure 4.4 - Approximated partial plane of the QUADRATIC's input space showing AV heuristic moves on node 5

On node 6, the AV could not find a solution. The location and size of node 6's domain (section 4.2.1), makes it considerably harder to find a solution than node 4 or 7. Figure 4.5 shows the AV operating on node 6. The heuristic start point is node 5's solution point. A *heuristic start point* is the first input point a heuristic uses to generate from. Modifying variable A, the AV was drawn to the notional boundary where many points were generated but no solution found. Figure 4.5 clearly shows the "homing-in" behaviour of the AV. When the AV generates a point that crosses the boundary from the (D>0) domain to the (D<0) domain, an upper-deviation is produced. In response, the AV starts an exploratory search from the sibling-traversal point it has found, which is closest to the boundary. This leads to a pattern search and the generation of a point that crosses the boundary, completing the cycle.

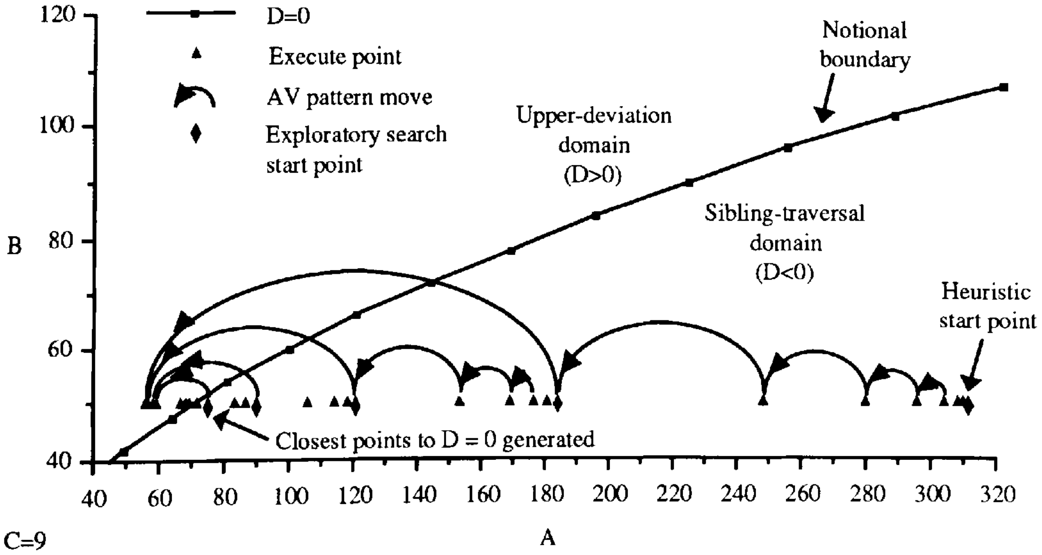


Figure 4.5 - Approximated partial plane of the QUADRATIC's input space showing the AV's moves on node 6

Iter range	Number of iterations :							Iteration producing closest point
	Variable modified			Improved sib trav	Degraded sib trav	Upper-deviation	Duplicate data	
	A	B	C					
1-10	10	0	0	8	1	1	0	10
11-20	10	0	0	9	0	1	0	20
21-30	10	0	0	7	0	3	0	29
31-40	6	2	2	2	3	5	0	33
41-50	4	4	2	0	5	5	10	33
Total	40	6	4	26	9	15	10	

Table 4.5 - Progression of the AV considering node 6

Table 4.5 breaks down the AV's progress on node 6. In the first 30 iterations progress was good, despite overstepping the boundary 5 times and is shown by the high number of iterations that produced a closer point. A point close to the boundary was found (72, 50, 9), solely by modifying input variable A. In the next 10 iterations progress turned around. On iteration 33, the AV produced the closest point of the 50 iterations. However, the AV cannot determine this, so continues. Indications that progress to a solution had ceased are shown in the last 20 iterations. The AV cycles around the three input variables, modifying each, without finding a closer point. Compared to the first 30 iterations, many more upper-deviations are produced. On each of the last 10 iterations, duplicate data was generated.

Viewing the QUADRATIC's input space in the locality of a heuristic's search helps to explain why the heuristic behaved in a particular way. Figures 4.6 and 4.7 show predicate values for nodes 6 and 7 in the neighbourhood of the closest point found (70, 50, 9) in iteration 33. In the figures an asterisk (*) represents a point that causes an upper-deviation, in this case to node 5. The AV's objective is to locate values for variables A, B and C, where D=0. Modifying variable A locates the closest point the AV can find, although the AV is not aware of this. Exploratory searches, from this closest point, modifying variables B and C, produce no improvement. The AV then cycles around all three variables again fruitlessly, until the node iteration threshold is met.

	52	*	*	*	*	*	*	*
B	51	*	*	*	*	*	*	-27
	50	*	*	*	-20	-56	-92	-128
	49	-11	-47	-83	-119	-155	-191	-227
		67	68	69	70	71	72	73
C=9					A			

Figure 4.6 - Run Q1 (A, B) partial input plane : AV considering node 6

	11	-679	-580	-479	-376
	10	-399	-300	-199	-96
C	9	-119	-20	*	*
	8	*	*	*	*
	7	*	*	*	*
		49	50	51	52
A=70			B		

Figure 4.7 - Run Q1 (B, C) partial input plane : AV considering node 6

Figures 4.6 and 4.7 show that no solution point (a predicate value of 0) exists in the nearest neighbourhood of the closest point (70, 50, 9) and there is no closer point. The nearest solution to where the AV was searching, exists at (64, 48, 9) and (81, 54, 9), if C is held constant at 9. To locate these would require the AV to modify both A and B at the same time. This would be a two dimensional search rather than the present single dimension search.

4.3.3 Run Q2

Table 4.6 shows this run's traversal results. This run's initial point caused the traversal of an alternative initial path to run Q1. Accordingly, the HATS harness considers different nodes from run Q1. Six of the seven nodes were traversed. Table 4.7 breaks down the AV's progress on node 6. No positive progress was made in the last 40 iterations. This is indicated by no improved sibling-traversals, an equal number of degraded sibling-traversals and upper-deviations, and duplicate data generated on every iteration.

Initial point path		1,3,5,7	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	1	
4	AV	5	
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
6	AV	50	

Table 4.6 - Run Q2 traversal results

The closest point the AV can find (4, 27, 46) is located on the fifth iteration. After this point has been located the AV cycles around the input variables making no positive progress. This can be seen from the number of iterations each variable was modified. This run demonstrates the AV's behaviour when the initial point is close to a notional boundary and no solution is found.

Iter range	Number of iterations :							Iteration producing closest point
	Variable modified			Improved sib trav	Degraded sib trav	Upper-deviation	Duplicate data	
	A	B	C					
1-10	8	2	0	4	2	4	2	5
11-20	4	2	4	0	5	5	10	
21-30	2	4	4	0	5	5	10	
31-40	4	4	2	0	5	5	10	
41-50	4	2	4	0	5	5	10	
Total	22	14	14	4	22	24	42	

Table 4.7 - Progression of the AV considering node 6

4.3.4 Run Q3

The traversal results for this run are shown in table 4.8. Only node 6 was untraversed.

Initial point path		1,3,4	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	1	
5	AV	10	7
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
6	AV	50	

Table 4.8 - Run Q3 traversal results

On node 6, the AV makes positive progress modifying variable A in the first 40 iterations (table 4.9). This is indicated by the high number of improved sibling-traversals. The few upper-deviations are due to the AV "homing-in" on the notional boundary and overstepping it, causing upper-deviations, as shown in figure 4.5.
In the last 20 iterations progress turns around and is indicated by the high number of iterations where upper-deviations are produced and duplicate data is generated. The closest point the AV can find is located in iteration 41 and from here on the AV cycles around the input variables.

Iter range	Number of iterations :							Iteration producing closest point
	Variable modified			Improved sib trav	Degraded sib trav	Upper-deviation	Duplicate data	
	A	B	C					
1-10	10	0	0	9	0	1	0	10
11-20	10	0	0	9	0	1	3	20
21-30	10	0	0	7	0	3	0	29
31-40	6	4	0	3	2	5	5	37
41-50	2	2	6	1	4	5	5	41
Total	38	6	6	29	6	15	13	

Table 4.9 - Progression of the AV considering node 6

4.3.5 Run Q4

The traversal results for this run are shown in table 4.10. Only node 6 was untraversed.

Initial point path		1,3,5,7	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	1	
4	AV	11	
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
6	AV	50	

Table 4.10 - Run Q4 traversal results

On node 6, the AV makes positive progress in the first 40 iterations (table 4.11), locating the closest point it can find in iteration 41. Indications that positive progress had ceased are shown in the last 10 iterations.

Iter	Number of iterations :							Iteration
range	Variable modified			Improved	Degraded	Upper-	Duplicate	producing
	A	B	C	sib trav	sib trav	deviation	data	closest point
1-10	10	0	0	9	0	1	0	10
11-20	10	0	0	8	0	2	0	20
21-30	10	0	0	5	1	4	4	27
31-40	0	2	8	5	2	3	0	39
41-50	2	2	6	1	4	5	9	41
Total	32	4	14	28	7	15	13	

Table 4.11 - Progression of the AV considering node 6

4.3.6 Branch Testing Discussion

Both DA and AV were used in four separate branch testing runs of the QUADRATIC procedure. The DA worked without fault in all occasions. However, there was little potential for any fault. There could not be any upper-deviation since nodes 2 and 3, which the DA is applicable to, are the immediate successors of the root node.

The application of the AV was both good and bad. With nodes 4 and 5, a solution was found in every run, modifying only one input variable. There was no difficulty in locating a solution point, due to the large size of the nodes’domains (section 4.2.1).

An observation made whilst the AV considered nodes 4 and 5 is that the number of iterations required to locate a solution is related to the predicate value distance between the heuristic's start point and the solution domain's expected notional boundary. *Predicate value distance* is the difference between the predicate value of two input points. In this case the first input point is the heuristic start point and the second is an input point that produces a predicate value of 0, which may or may not exist in the test procedures input space.

Run	Node	Predicate value distance	Iters
Q2	4	4679	5
Q1	5	38200	9
Q3	5	77641	10
Q4	4	210501	11

Table 4.12 - Relationship between the predicate value distance to an expected solution and the number of iterations taken to locate the solution

Table 4.12 clearly shows that as the distance increases so does the number of iterations. The effect of the ever-increasing AV pattern search steps is shown. The number of iterations increase linearly as the predicate value distance increases exponentially.

With node 6, the AV did not locate a solution point in any of the runs. Once the AV had located the (D=0) notional boundary, the same behaviour was observed in all runs. Upper-deviations, degraded sibling-traversals and duplicate data generations all increase. The positive progress exhibited in runs Q1, Q3 and Q4, is the AV locating the notional boundary. The number of iterations positive progress takes place for is related to the predicate value distance from the heuristic's start point to the (D=0) notional boundary.

4.3.7 Comparison of HATS with Random Testing

It is desirable for any automated test data generator to compare favourably to random testing. To compare them it is necessary for both to be conducted under the same conditions. Both aim to satisfy the all branches criterion on the QUADRATIC procedure. The range of values HATS's initial point can be selected in and random testing can generate is constrained to ±1000 for each input variable.

Each random testing run continues until all nodes are traversed. The number of iterations taken to traverse a node is recorded. This is conducted for 500 runs and the average for each node calculated.

Node	HATS iterations	Random iterations
1	IPT	1
2	1	2160
3	IPT	1
4	5-11	1.618
5	9-10	2.728
6	NNT	572941
7	IPT/CT	2.728

Key :

IPT	Initial point traversed
NNT	Node not traversed
CT	Coincidentally traversed

Table 4.13 - The number of iterations taken by HATS and random testing for each of the QUADRATIC's nodes

Table 4.13 shows the iterations taken by HATS and random testing for each node in the QUADRATIC. The results shown for HATS are varied. When a node is considered and traversed more than once, a minimum and maximum number of iterations is given. If a node is traversed by the initial point or coincidentally, this is stated as the node was not considered by the HATS harness.

HATS took significantly less iterations than random on node 2, since an equality predicate was involved (A=0) and HATS was able to use the DA. Nodes 1, 3 and 7 compare equally since they all have large satisfying input domains which are easy to hit. HATS is slightly worse than random on nodes 4 and 5. Both these nodes have large input domains which are easy to hit. However, the AV uses those iterations to move from one node's domain to the other. The number of iterations used depends on how far the heuristic start point is from the solution domain's boundary (predicate value distance). This would have applied to node 7 had HATS considered it. With node 6 HATS was unsuccessful, however, random took a significantly large number of iterations to find a solution.

4.4 Mutation Analysis

4.4.1 The Appeal of Mutation Analysis

Mutation Analysis (MA) (section 2.3; Budd, 1981) of the QUADRATIC provides a double benefit. First, MA provides several levels of analysis. Some of which are harder to satisfy than branch testing. Second, we can observe how the HATS harness, in particular the heuristics, react to MA, enabling further evaluation and improvement.

MA possesses a number of desirable features. The levels of MA equate to several structural testing criteria and do not require significant modification or further development of HATS. So far the AV has been unable to generate test data to traverse node 6. MA may shed further light on this problem and, possibly, other problems.

4.4.2 Using Mutation Analysis with HATS

There are two main differences between the way MA is traditionally practised and the way it will be practised in the following experiments. Firstly, a mutant revealing point generated in one HATS run will not be carried forward and executed before subsequent HATS runs. A single point will start the HATS run making it the responsibility of HATS to locate a mutant revealing point. Secondly, rather than comparing output from the original and mutated procedures, output value checking postconditions have been incorporated into the mutated QUADRATIC. If the postconditions show that the input values and output values disagree then the execute point has revealed the mutant. This check is performed directly after the QUADRATIC has executed. Hence there is no need to execute the original QUADRATIC after every HATS iteration.

For each mutation the original QUADRATIC is changed and recompiled manually. The mutated QUADRATIC then executes upon a single, initial point. The HATS harness then attempts to cause every branch in the QUADRATIC to be traversed, as previously. The HATS harness has no knowledge of the QUADRATIC's mutations. If the postconditions are violated, the HATS harness terminates and reports that the mutant has been revealed. Otherwise, the system continues until either all nodes are traversed or are deemed infeasible.

Mutant	Mutation operator	Line no	Original condition	Mutated condition	Equivalent
QM1	Stmt del	6			N
QM2	"	9			N
QM3	"	11			N
QM4	"	13			N
QM5	"	19 - 31			N
QM6	"	20			N
QM7	"	25			N
QM8	Cond rep	8	(A=0)	(TRUE)	N
QM9	"	8	"	(FALSE)	N
QM10	"	12	(D>0)	(TRUE)	N
QM11	"	12	"	(FALSE)	N
QM12	"	19	(D=0)	(TRUE)	N
QM13	"	19	"	(FALSE)	N
QM40	Stmt del	7			N
QM41	"	8 - 33			N
QM42	"	12 - 32			N
QM43	"	14			N
QM44	"	15			N
QM45	"	16			N
QM46	"	17			N
QM47	"	21			N
QM48	"	22			N
QM49	"	23			N
QM50	"	26			N
QM51	"	27			N
QM52	"	28			N
QM53	"	29			N
QM54	"	30			N

Key :	
Stmt del	Statement deletion
Cond rep	Condition replacement

Table 4.14 - QUADRATIC's first round statement analysis mutants

4.4.3 Mutants Produced

The first two levels of MA, statement and predicate analysis, have been used. Predicate analysis is harder to satisfy than branch testing. Table 4.14 and 4.15 contain the mutants produced for statement and predicate analysis respectively. In total there are 52 mutants produced, of which 49 are nonequivalent. Statement analysis produced 28 mutants, 22 from statement deletion and 6 from condition replacement. None are equivalent. Predicate analysis produced 24 mutants, 6 from predicate alteration by a small value, 3 from absolute operator insertion (2 equivalent) and 15 from relational operator alteration (1 equivalent).

Mutant	Mutation operator	Line no	Original condition	Mutated condition	Equivalent
QM14	Pred alt	8	(A=0)	(A=1)	N
QM15	"	8	"	(A=-1)	N
QM16	"	12	(D>0)	(D>1)	N
QM17	"	12	"	(D>-1)	N
QM18	"	19	(D=0)	(D=1)	N
QM19	"	19	"	(D=-1)	N
QM20	Abs op ins	12	(D>0)	(abs(D)>0)	N
QM21	"	19	(D=0)	(abs(D)=0)	Y
QM22	Rel op alt	8	(A=0)	(A/=0)	N
QM23	"	8	"	(A>0)	N
QM24	"	8	"	(A>=0)	N
QM25	"	8	"	(A<0)	N
QM26	"	8	"	(A<=0)	N
QM27	"	12	(D>0)	(D<=0)	N
QM28	"	12	"	(D<0)	N
QM29	"	12	"	(D/=0)	N
QM30	"	12	"	(D=0)	N
QM31	"	12	"	(D>=0)	N
QM32	"	19	(D=0)	(D/=0)	N
QM33	"	19	"	(D>0)	N
QM34	"	19	"	(D>=0)	Y
QM35	"	19	"	(D<0)	N
QM36	"	19	"	(D<=0)	N
QM60	Abs op ins	8	(A=0)	(abs(A)=0)	Y

Key	
Pred alt	Predicate alteration
Abs op ins	Absolute operator insertion
Rel op alt	Relational operator alteration

Table 4.15 - QUADRATIC's first round predicate analysis mutants

4.5 The First Round of Mutation Analysis

4.5.1 Experimental Set-up

The initial point for each HATS run has been randomly generated, in the arbitrary range 0 to 1000, for each input variable. The node iteration threshold is 50. Only the 49 nonequivalent mutants are the subjects of the HATS harness.

4.5.2 Statement Analysis Results and Discussion

All 28 nonequivalent mutants were used. Table 4.16 shows that 13 (46%) statement analysis mutants were revealed by the initial point. Mutant QM41 is revealed by any input value and mutants QM3, QM8 and QM42 are revealed by any point that has a non zero value for A. The remaining mutants were revealed with a point from one of the two largest input domains, ((A<>0) & (D>0)) and ((A<>0) & (D<0)). In this case the initial point happened to be in the mutants revealing domain. However, some easy to reveal mutants will not be revealed if the initial point is not in the mutants revealing domain.

Mutant	Initial point		
	A	B	C
QM3	153	76	478
QM4	208	611	352
QM5	343	190	561
QM7	63	327	889
QM8	756	752	654
QM41	941	95	979
QM42	432	294	753
QM43	51	96	905
QM45	6	779	98
QM46	808	939	202
QM50	981	462	293
QM51	801	478	544
QM53	300	232	903
Number of mutants			13

Table 4.16 - First round statement analysis mutants revealed by the initial point

Table 4.17 shows that 10 (36%) statement analysis mutants were revealed by the DA and AV. DA revealed four mutants and AV, six. One would have expected the initial point to reveal QM1 and QM40, since they are in the linear statement sequence to the first branch and are executed every time. However, all paths, except path 1, 2 define the variables X1 and X2 which are initialised by the two deleted statements, so do not reveal the mutants. To reveal the mutants, path 1, 2 must be taken.

The AV revealed mutants by modifying an initial point, which was in one of the two largest domains, to the other largest domain.

Mutant	Initial point			Heuristic	Considered	No iters	Revealing point		
	A	B	C		node	on node	A	B	C
QM1	227	262	26	DA	2	1	0	262	26
QM2	253	798	746	DA	2	1	0	798	746
QM9	458	154	291	DA	2	1	0	154	291
QM10	143	845	369	AV	5	9	654	845	369
QM11	761	291	768	AV	4	11	-262	291	768
QM12	80	873	264	AV	5	10	1103	873	264
QM40	249	763	713	DA	2	1	0	763	713
QM44	602	663	792	AV	4	10	91	663	792
QM52	187	919	890	AV	5	6	250	919	890
QM54	5	760	649	AV	5	8	260	760	649
Number of mutants			10						

Table 4.17 - First round statement analysis mutants revealed by a heuristic

Nevertheless 5 (18%) statement analysis mutants were unrevealed (table 4.18). Each of these mutants required a point that made program variable D = 0. This further confirms previous observations (section 4.3).

With each of the unrevealed mutants, the AV was considering node 6 and the node iteration threshold was reached. The AV exhibited the same behaviour here to that of the branch experiments after locating the notional boundary and being unable to locate a solution.

Mutant	Initial point		
	A	B	C
QM6	561	749	414
QM13	801	319	363
QM47	337	907	168
QM48	743	884	589
QM49	322	517	102
Number of mutants			5

Table 4.18 - First round statement analysis mutants unrevealed by the initial point or heuristics

4.5.3 Predicate Analysis Results and Discussion

Of the 21 nonequivalent mutants, 9 (42%) were revealed by the initial point (table 4.19). The revealing domains for all of these mutants is very large. QM22 is revealed with any non-zero value for variable A; QM23 and QM24 with variable A greater than zero; QM20, QM27, QM28, QM32, QM35 and QM36 with any point that gives variable D a value less than zero. The range 0 to 1000 for the generation of the initial points prevented QM25 and QM26 from potentially being revealed since the mutants require variable A to be less than zero.

Mutant	Initial point		
	A	B	C
QM20	355	868	992
QM22	252	608	493
QM23	29	578	3
QM24	60	338	573
QM27	700	540	706
QM28	428	492	476
QM32	282	436	265
QM35	885	331	257
QM36	99	591	888
Number of mutants			9

Table 4.19 - First round predicate analysis mutants revealed by the initial point

Six (29%) predicate analysis mutants were revealed by the heuristics (table 4.20). The three mutants revealed by the DA required specific values of variable A. Of the three mutants AV revealed, QM29 and QM30 were revealed in the same manner as the AV revealed statement analysis mutants. However, QM31 requires D=0, to be revealed. The AV has located a point which makes D=0. The AV achieved this by first reducing variable A to one. No further improvement could be made modifying variable A so B was considered and an even number found where no further improvement could be made modifying B. Now modifying C with A=1 and B even, gave the smallest possible change in predicate value; just 4. But much more importantly a solution lay in variable C's dimension, and was located.

Mutant	Initial point			Heuristic	Considered	No iters	Revealing point		
	A	B	C		node	on node	A	B	C
QM14	713	111	70	DA	2	1	1	111	70
QM15	105	770	969	DA	2	1	-1	770	969
QM25	631	325	997	DA	2	1	-1	325	997
QM29	88	522	408	AV	5	7	215	522	408
QM30	973	979	361	AV	4	10	462	979	361
QM31	322	31	586	AV	6	30	1	48	576
Number of mutants			6						

Table 4.20 - First round predicate analysis mutants revealed by a heuristic

Six (29%) predicate analysis mutants were unrevealed by the heuristics (table 4.21). To be revealed, QM26 requires a negative value for variable A. When the DA considered QM26 a value of 0 was generated for A since the predicate (A<= 0) involved an equality. Consequently, node 2 was traversed and the mutant unrevealed. The remaining unrevealed mutants require variable D to have specific values (-1, 0 or 1) to be revealed. Experience has shown that locating a point to give D a value of 0 is very difficult and rarely happens. It follows that locating a point to give D other single values is also difficult.

Mutant	Initial point		
	A	B	C
QM16	421	821	334
QM17	104	814	289
QM18	252	469	652
QM19	654	109	554
QM26	71	781	920
QM33	427	188	608
Number of mutants			6

Table 4.21 - First round predicate analysis mutants unrevealed by the initial point or heuristics

4.5.4 Mutation Analysis Summary

Table 4.22 shows that of the 49 mutants used, 38 (78%) were revealed, 22 (45%) by the initial test point and 16 (33%) by the DA and AV heuristics. Eleven mutants (22%) were unrevealed. Offutt (1992) states that creating a test data set which has a mutation score greater than 95%, is difficult, but effective at finding faults. Therefore we would wish to improve upon these results, in an attempt to achieve a score closer to 95% or better.

Many of the mutants have large revealing domains. To produce a revealing point was not difficult and many mutants (45%) were revealed by the initial point without the use of the HATS harness. All the mutants revealed by an initial point would be easily revealed by a heuristic. Mutants remain unrevealed because the heuristics did not produce points that caused specific values in variables.

MA Level	Mutation operator	No of mutants used	Number mutants revealed :			No of mutants unrevealed
			total	by initial test point	by heuristic	
Statement analysis	Stmt del	22	18	12	6	4
	Cond rep	6	5	1	4	1
	Overall	28	23	13	10	5
Predicate analysis	Pred Alt	6	2	0	2	4
	Abs op ins	1	1	1	0	0
	Rel op alt	14	12	8	4	2
	Overall	21	15	9	6	6
	MA overall	49	38	22	16	11

Table 4.22 - Overall mutation analysis results for the QUADRATIC

4.6 Heuristic Discussion

This section discusses the performance of the DA and AV for both branch testing and MA of the QUADRATIC.

4.6.1 DA Discussion

The DA caused traversal of every node it considered, and it significantly outperforms random testing when an equality predicate is involved.

4.6.2 AV Discussion

The AV located solution points where there is a large solution domain for a node. In comparison with random, the AV is marginally worse. Further, the AV rarely located a solution when the solution domain is small and sparsely located. The QUADRATIC predicate (D = 0), or mutations of, revealed this. On virtually all the AV's considerations of this predicate, the notional boundary is located, but a solution is not. Once the notional boundary is located and all the input variables have been modified there is no further improvement of predicate value. Here the AV has effectively become stuck on the closest point it can find to the notional boundary.
The single run where the AV located a point that made D=0 was enabled through the initial point. It was not due to any unusual operation of the AV. Had some other initial point been generated then the AV would have more than likely failed.
The reasons that the AV is generally unable to locate a solution point for node 6 appear to lie with the point closeness metric (section 3.5.5) and the nature of the AV. Ideally by minimising the predicate value for a node the AV will locate a point on the opposite side of a domain boundary or directly upon it. On doing so the considered node should be traversed. This is a belief also shared by Prather and Myers (1987). However in the QUADRATIC's case, the point closeness metric leads the AV to a notional boundary where there is generally no solutions to node 6. The AV's

deterministic, localised and single-dimension search nature do not help to overcome the misguidance of the point closeness metric.

Solutions to node 6 do exist. They are located along the notional boundary. A proposed solution to the above problem involves using the notional boundary as a guide to a solution point. This proposal is developed into a new heuristic in the next chapter (5). A further concern is of the AV's behaviour as it approaches a boundary where points on the opposite side of the boundary cause upper-deviations (figure 4.5). This causes the AV to slow down its search and waste iterations. To reduce iterations the linear relationship between an input variable and a predicate variable can be exploited. Such a heuristic would perform linear extrapolation to exactly predict boundary value points and locate solution points.

5 The New Heuristics and Improved Quadratic Results

5.1 Introduction

The previous chapter indicated that results from using the DA and AV on the QUADRATIC need to be improved. Specifically, the AV has difficulties locating a solution that lies upon a notional boundary and it unnecessarily uses iterations. In an attempt to overcome these problems two further heuristics are proposed and their results presented (Holmes, et al, 1993).

5.2 The Linear Predictor Heuristic

The Linear Predictor (LP) exploits a linear relationship that may exist between an input variable and a predicate variable by extrapolating a boundary located point then modifying it to cross the boundary and traverse the considered node. In this way fewer iterations will be used than the AV.

A program study has indicated that in production data processing applications most predicates are linear and involve a small number of variables. White and Cohen (1980) state, from a study conducted by Cohen (1978) on 50 COBOL programs consisting of 1225 predicates, that 77.1% of the predicates involved one variable and 10.2% of the predicates involved two variables; 87.3% of the predicates were linear, only one predicate was nonlinear and the remaining 12.6% of the predicates were input-independent.

Considering 120 production PL/1 programs, Elshoff (1976) discovered that 98% of expressions had less than two operators and the occurrence of the arithmetic operators, +, -, *, /, in expressions were 68.7%, 16.2%, 8.9% and 2.8% respectively. In addition, Elshoff states "... plus one accounts for many of the operations". Knuth's (1971) study of FORTRAN programs also agrees, stating that 40% of additions are plus ones. He also states that 86% of assignment statements are of the form A = B, A = B + C or A = B - C. These studies indicate that many predicates will be simple and their interpretations will be linear, hence the LP has potentially a wide applicability.

5.2.1 Overview and Phases of the Linear Predictor

Figure 5.1 illustrates the LP operating in an ideal situation. The figure represents the partial input plane for an arbitrary procedure with two integer input variables, X and Y, and a condition dependent upon both these variables. Points directly upon the boundary and to the left of it cause sibling-traversal. Points to the right of the boundary cause considered node traversal.

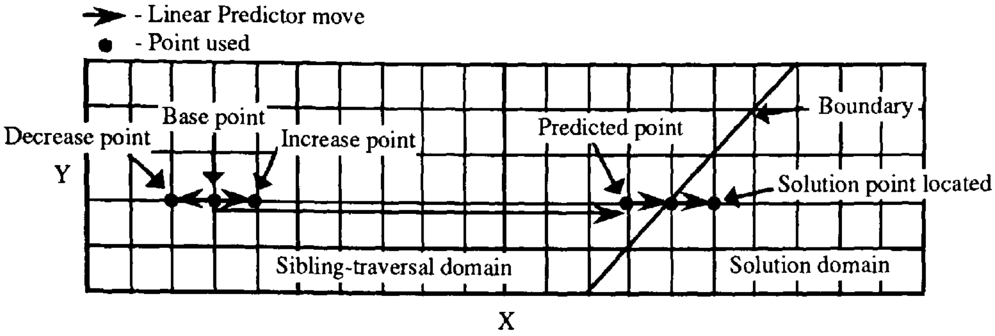


Figure 5.1 - Operation of the Linear Predictor

The LP has three phases. The first phase, *Determine-linearity* (DL) determines the linearity of the considered input variable with respect to the considered predicate. A base point is used to produce the increase and decrease points. Figure 5.1 shows that variable X is modified to produce these two points, while Y remains constant. Using the increase, decrease and base points the Determine-linearity phase can make a decision on the linearity of the predicate.

If linear, a value is extrapolated for the considered input variable X, that should be very near or directly on the expected boundary. Before this extrapolated point is executed, it is modified slightly away from the expected boundary toward the base point, so that later, boundary spanning points can be generated. This modified point is termed the *Predicted point*. This forms the preparation for the next phase, Predictor.

If the considered input variable is nonlinear with respect to the considered predicate then preparation is made for the Creeper phase by setting the modification direction according to the increase or decrease point that came closest to the expected boundary.

The *Predictor* phase modifies the Predicted point to produce further points that cross the boundary and cause the considered node to be traversed. Figure 5.1 shows the Predictor phase modifying the Predicted point to a point directly on the boundary, which causes sibling-traversal. This point is then modified to a point that causes considered node traversal. Boundary located points have been generated.

The *Creeper* phase modifies the considered input variable by small steps in a direction toward the expected boundary.

5.2.2 Components and Functions of the Linear Predictor

The LP's pseudo code is in Appendix A3.3.

5.2.2.1 First Iteration Set-up, Generator and Evaluator

The first iteration set-up component specifies the current phase as Determine-linearity, sets the DL base point to a sibling-traversal point that has the closest predicate value to the expected boundary and specifies that the first input variable shall be considered and increased by one.

The generator component applies the input point generation instructions to the generate point producing the execute point which the test software will execute upon.

The evaluator component is called after the test procedure has executed on the execute point. If the LP has been successful, control returns to the HATS harness. Otherwise, evaluation is performed dependent upon the LP's phase, traversal-effect of the execute point and, with the Predictor phase, the closeness of the execute point to the expected boundary.

5.2.2.2 Determine-linearity Phase

The Determine-linearity phase establishes the linearity of an input variable with respect to the considered predicate and prepares for either the Predictor or Creeper phases. The following example illustrates the DL phase in some detail.

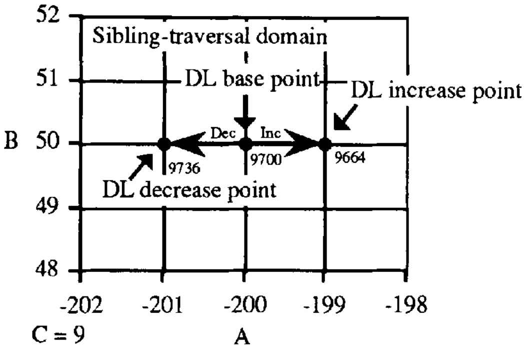


Figure 5.2 - (A, B) partial input plane : LP in the Determine-linearity phase considering node 5 of the QUADRATIC

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
BP				-200	50	9	ST	9700	+A
1	-200	50	9	-199	50	9	ST	9664	-A
2	-200	50	9	-201	50	9	ST	9736	PP

Table 5.1 - HATS run excerpt : LP in the Determine-linearity phase considering node 5 of the QUADRATIC and modifying input variable A

In the previous chapter, the AV considered node 5 of run Q1. The example illustrates the LP on the same problem. As only the initial point (-200, 50, 9) has caused sibling-traversal, it is selected as the DL base point. Using the DL base point, the first variable, A, is increased and decreased by 1, producing the DL increase and DL decrease points. Both these points are executed directly after they are generated. Figure 5.2 shows these moves and table 5.1, the LP's operation.

The LP now decides if the considered predicate (D <= 0) is linear with respect to input variable A. The increase and decrease points are used to determine that a linear relationship does exist. Thus a value of 69.44 is extrapolated for A. The fraction is truncated, giving 69, since the target data type is integer. If the point (69, 50, 9) were executed, it should have a predicate value of zero or very close to zero.

To ensure that boundary spanning points are generated, the Determine-linearity phase instructs the first generation of the next phase, Predictor, to modify input variable A back toward the DL base point, producing the Predicted point (68, 50, 9). This example continues in the next section (5.2.2.3).

The operation of the Determine-linearity phase is now described. Ideally sibling-traversal will result from execution of both the increase and decrease points so that the linearity of the considered input variable on the considered predicate can be established. If the considered input variable is influential then an attempt is made to determine its linearity, otherwise the next variable is considered.

Determining if the considered predicate is linear with respect to the considered input variable, is achieved by comparing the difference between the DL base point's predicate value and the predicate values of the increase and decrease points. If they are the same then the considered predicate is deemed linear and the Predictor phase is prepared for, otherwise the predicate is deemed nonlinear and the Creeper phase is prepared for.

Preparation for the Predictor phase involves an extrapolation and modification. The extrapolation uses the considered input variable values and predicate values of the DL base and increase points. The predicate value of a point that lies directly on the expected boundary, normally zero, is used as the target. Modifying the extrapolated value back toward the DL base point's value by one, ensures that a point adjacent to the expected boundary, causing sibling-traversal is produced. The Predicted point is formed from the modified extrapolated value together with the remaining input variable values from the DL base point. Modifying this point across the boundary produces boundary spanning test data.

If an upper-deviation occurs in the Determine-linearity phase then the DL base point is close to a boundary with respect to the considered input variable. Abandoning the considered input variable and considering another, may locate a point even closer to the boundary or a solution.

Preparation for the Creeper phase involves setting the modification direction from the DL increase or decrease point that came closest to the expected boundary.

5.2.2.3 Predictor Phase

The Predictor phase is active from the generation of the Predicted point onwards whilst considering the same input variable. The following example illustrates the Predictor phase in some detail and continues from the example given in the previous section (5.2.2.2).

This phase commences with the generation of the Predicted point (68, 50, 9). Executing the Predicted point ideally causes the sibling node (4) to be traversed and is close to the boundary. This takes place and is shown figure 5.3 and table 5.2.

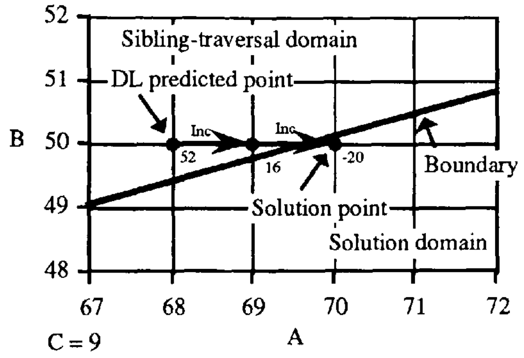


Figure 5.3 - (A, B) partial input plane : LP in the Predictor phase considering node 5 of the QUADRATIC

The modification direction is now reversed so that the next point generated is closer to the boundary, on to it or over it. Thus for the fourth iteration the modification direction is reversed from decrease to increase, and variable A is modified by 1. Execution produces a sibling-traversal with a predicate value closer to zero than the Predicted point. This indicates progress is being made toward the boundary. Hence for iteration 5, the Predictor maintains the present direction of increase and step size of 1. The fifth iteration causes the considered node (5), to be traversed.

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
3	69	50	9	68	50	9	ST	52	+A
4	68	50	9	69	50	9	ST	16	+A
5	69	50	9	70	50	9	NT	-20	SUCC

Table 5.2 - HATS run excerpt : LP in the Predictor phase considering node 5 of the QUADRATIC and modifying input variable A

Applying the AV to node 5 from the base point (-200, 50, 9) took 9 iterations to find a solution point (section 4.3.2). However, the LP took only 5 iterations and generated boundary located points.

The operation of the Predictor phase is now described. Following the execution of the Predicted point and all subsequent points in the Predictor phase, evaluators are invoked based on each point's traversal-effect and its closeness to the expected boundary.

When the execute point causes sibling-traversal and is closer to the expected boundary than the closest point found two checks are made. The first is if all the points from the Predicted point onwards caused upper-deviations. This indicates that the Predicted point entered an interfering predicate's domain and its boundary has been passed through. The considered input variable is abandoned, with the view that a solution may be located from the closest point found modifying some other input variable. The second check is if the Predicted point has just executed. If so, the modification direction is reversed and the step size remains at one for the next generation. Otherwise, execution of the Predicted point has passed and progress is being made toward the boundary using a succession of small steps, so both the direction and step size are unchanged for the next generation. It is anticipated that, when there are no interfering predicates, the considered node should be traversed within two iterations from the execution of the Predicted point.

When the execute point causes a sibling-traversal and is further from the expected boundary than the closest point found, the considered input variable is abandoned. Although progress toward the expected boundary may have been made, this has ceased and to continue modifying the considered input variable would more than likely be worthless. Modifying another input variable from the closest point found may locate a solution or closer point.

When the execute point causes an upper-deviation two checks are made. The first, checks if all the points from the Predicted point onwards have caused closer sibling-traversals. This situation indicates that an interfering predicate has just been encountered. The considered input variable is abandoned on the basis that a solution may be located modifying another input variable from the closest point found.

The second, checks if an upper-deviation occurred on executing the Predicted point. If so, there is an interfering predicate. Rather than abandoning the considered variable, the Predicted point may be close to a solution point or point closer to the expected boundary. Instead of reversing the modification direction toward the expected boundary, the direction is left unchanged so that points generated move away from the expected boundary and out of the interfering predicates domain. If upper-deviations continue to occur after a small number of iterations then the considered input variable is abandoned.

5.2.2.4 Creeper Phase

The Creeper phase is active from the execution of the first creep point onwards whilst considering the same input variable. In the Determine-linearity phase example (section 5.2.2.2), input variable A was considered from the point (-200, 50, 9). The following example illustrates how the Creeper phase operates considering input variable B from the same point.

Each time a new input variable is considered the Determine-linearity phase is used first. From figure 5.4 we can calculate that the increase point has a difference in predicate value with the base point of 101 and the decrease point has a difference of 99. Since the differences are not equal, variable B is deemed non-linear with respect to the considered predicate (D <= 0).

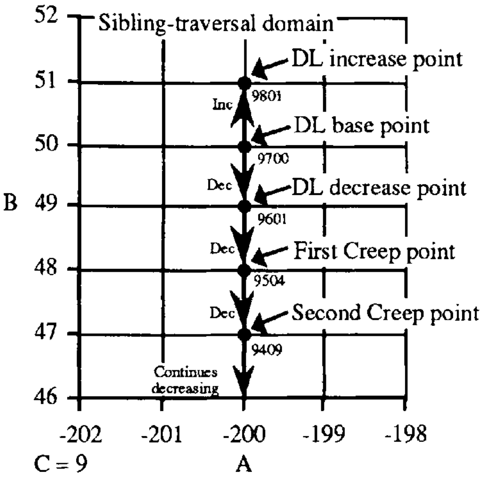


Figure 5.4 - (A, B) partial input plane : LP in the Determine-linearity and Creeper phases, considering node 5 of the QUADRATIC

Now the Determine-linearity phase prepares for the Creeper phase. The generate point for the Creeper is the DL decrease point, as it is the closest point to the expected boundary, and the modification direction is decrease. The step size is set to one, since the Creeper takes small steps toward the expected boundary.

Execution of the first Creeper phase point produces the predicate value 9504 which is closer to the expected boundary than the DL decrease point. The second and subsequent iterations continue to decrease input variable B by 1. If a point is generated that causes traversal of the considered node then the LP has succeeded. However, the considered variable is abandoned if the execute point is further away from the expected boundary or causes an upper-deviation.

Following the execution of a Creeper point, evaluators are invoked based on the traversal-effect of the execute point. When the execute point causes a sibling-traversal and is closer to the expected boundary than the closest point found, then progress is being made and the input point generation instructions are not changed. However, if the execute point is further from the expected boundary then progress has stopped and to continue further would more than likely be worthless. Nevertheless, a closer point may have been located and further progress may be made by considering another input variable from the closest point found.

When the execute point encounters an upper-deviation this indicates that there is an interfering predicate and its boundary has just been crossed. To modify the considered variable further would more than likely be worthless, so the considered input variable is abandoned for another. Nevertheless, a point closer to the boundary has been located.

5.2.2.5 Abandoning Consideration of an Input Variable

When the considered input variable is abandoned the following takes place. If the termination criteria (section 5.2.2.6) are met, the LP terminates. Hence, the LP has been unable to locate a solution point.

If the termination criteria are not met, the Determine-linearity phase is prepared for on the next input variable. The new considered input variable is the next in-line to the old one or the first, if the old one was the last. The closest sibling-traversal point to the considered predicate's boundary is searched for and becomes the DL base point.

5.2.2.6 Terminator

In chapter 4 we saw how the AV cycled round the input variables trying to find a closer point or a solution. Generally, neither of these were found and many iterations were wasted. To overcome this the LP has adopted a "law of diminishing returns" by incorporating termination criteria based on a threshold of unpromising effects. *Unpromising effects* are anything that indicate a turn around in progress has occurred, i.e. an upper-deviation or an execute point further from the expected boundary than the closest point found. The LP keeps a count of unpromising effects. Should this count reach a threshold, equal to the number of input variables, then the LP terminates. In other words, termination takes place after modifying each input variable produced an unpromising effect. When a promising effect takes place the unpromising effects count is reset to zero. An example of a promising effect is locating a point closer to the expected boundary than the closest point found.

The following example illustrates the LP's termination criteria by applying the heuristic to node 6 of the QUADRATIC. This closest sibling-traversal point (70, 50, 9) was produced as a solution to node 5 in the Predictor phase example (section 5.2.2.3). Let us assume this is the only point to traverse the path 1, 3, 5, 7. The unpromising effects threshold is 3, since there are three input variables.

Table 5.3 is a *HATS run excerpt* which contains pre and post test software execution data for a range of iterations for a HATS run. An excerpt includes, from left to right, the iteration of the heuristic on the node, which is always shown. The pre-execution data can include the heuristic used, the heuristic's phase and the input variable being modified. The generate point and execute point are always shown. Postexecution data can include the execute point's traversal effect and predicate value if the execute point caused sibling-traversal. Also shown can be the next action the heuristic takes and other data. The abbreviations used in a HATS run excerpt are defined in the glossary. The moves contained in table 5.3 are shown in figure 5.5.

Iter	Var	Generate point			Execute point			Trav	Pred	Next	UP eff
		A	B	C	A	B	C	effect	value	action	count
1	A	70	50	9	71	50	9	ST	-56	-A	0
2	A	70	50	9	69	50	9	UD		TNV	1
3	B	70	50	9	70	51	9	UD		TNV	2
4	C	70	50	9	70	50	10	ST	-300	-A	2
5	C	70	50	9	70	50	8	UD		TERM	3

Table 5.3 - HATS run excerpt of the LP in the Determine-linearity phase considering node 6 of the QUADRATIC, demonstrating the use of the unpromising effects count to minimise unnecessary iterations

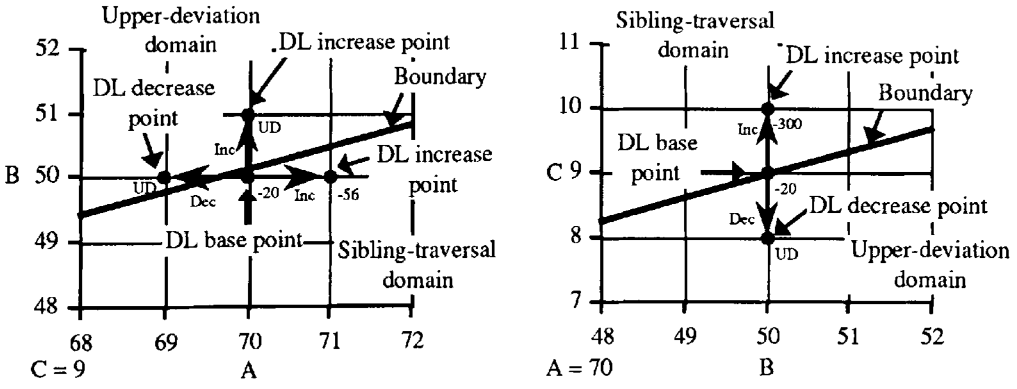


Figure 5.5 - (A, B) and (B, C) partial input planes : LP in the Determine-linearity phase, considering node 6 of the QUADRATIC

Since the base point is adjacent to a notional boundary, modifying it in both directions in any dimension will encounter upper-deviations. Such modifications are essential to the Determine-linearity phase. Upper-deviations occur in iterations 2, 3 and 5, causing the unpromising effects count to be incremented. When variable C is abandoned after iteration 5, the LP terminates since the unpromising effects count is equal to the number of input variables (3).

Table 5.4 shows the unpromising effects and table 5.5 the promising effects that the LP recognises. The threshold is reasonable since the heuristic will have cycled once round the input variables without improvement and to continue would more than likely be worthless.

Phase	Unpromising Effect
Determine-linearity	Input variable non-linear and neither increase or decrease points came closer to expected boundary
"	Upper-deviation encountered during execution of increase or decrease point
Predictor	Execute point, from Predicted point onwards, has predicate value further from the expected boundary than the closest point found
"	Execute point caused upper-deviation when points from Predicted point onwards caused sibling-traversal
"	Execute points, from the Predicted point onwards, caused upper-deviation and upper-deviation threshold has been reached
Creeper	Execute point has predicate value further from the expected boundary than the closest point found
"	Execute point caused upper-deviation

Table 5.4 - Events that cause the unpromising effects count to be incremented by one

Phase	Promising Effect
Determine-linearity	LP considers s new node
Predictor	Execute point caused sibling-traversal and is closer to expected boundary than closest point found and there has been no upper-deviations from the Predicted point onwards
Creep	Execute point caused sibling-traversal and is closer to the expected boundary than the closest point found

Table 5.5 - Events that cause the unpromising effects count to be reset to zero

5.3 The Boundary Follower Heuristic

When generating test data for an equality predicate whose two sides (sub-expressions) are of the integer data type and are some function of the input variables then this presents a problem known as the *notional boundary located point problem* . This problem has revealed its self through the AV's consideration of node 6 in the QUADRATIC.

The Boundary Follower (BF) uses a real or notional boundary as a guide in an attempt to locate solution points that lie on the boundary. While a boundary is being followed only boundary value test data (White and Cohen, 1980; Myers, 1979; Abbott, 1986) will be generated. This is a beneficial side-effect and such data has been shown to be better at revealing faults than data that does not explore the boundaries (i.e. branch coverage) (Myers, 1979; Basili and Selby, 1987).

In the field of constrained numerical optimisation (Gill and Murray, 1974; Box, et al, 1969), boundary or constraint following has been adopted by both gradient and direct search methods. Two gradient methods, *Riding the Constraint* and *Hemstitching*, due to Roberts and Lyvers (1961), use derivatives of the constraint function. Calculating derivatives for the test software is an additional overhead and may be very complex or not possible. Further, it would defeat the objective of not understanding the test software . The direct search method, *Pattern Search* (Hooke and Jeeves, 1961), has been extended to follow constraints (Klingman and Himmelblau, 1964; Glass and Cooper, 1965)

With nonlinear constraint functions, the above methods, may not accurately follow a constraint, only approximate to it. There is a possibility that some small parts of the constraint boundary may be missed through the method's normal operation or a speed up of the search. Normally, these methods would work with independent floating-point variables, not integer variables.

Adby and Dempster (1974) state "If the minimum lies on the constraint boundary then even the techniques of constrained optimisation may not work". This confirms the difficulty of this problem with, I suspect, floating-point variables. How well the above methods would operate on integer variables is a subject of further study.

5.3.1 Following Domain Boundaries in the Program Input Space

The test software may have many input domains. To follow a domain's boundary involves selecting a series of points that are either on or just off the boundary. The order they are selected in gives rise to a movement along the boundary in some direction. Examining partial input planes of the QUADRATIC led to the hypothesis that, to be certain a boundary is being accurately followed it must be crossed regularly. Crossing a boundary regularly ensures that it is not deviated from. To help the BF locate a solution, sibling-traversals must be regularly produced. The BF is a direct search technique that first locates a boundary then follows it. The BF follows a boundary in two dimensions, although a boundary may exist in a higher number of dimensions.

5.3.2 Overview of the Boundary Follower

Figure 5.6 illustrates the BF operating in an ideal situation. The figure represents a partial input plane for an arbitrary procedure with two integer input variables, X and Y, and a nonlinear condition, dependent upon both these variables. In the partial input plane (figure 5.6) there is one solution point which causes considered node traversal and all other points cause sibling-traversal.

The BF uses both variables X and Y to follow the boundary. To ensure that the heuristic is "sticking" to the boundary, modifying variable X must always cross the boundary. Modifying variable Y produces a point that stays on one side of the boundary. The modification of each variable alternates so that the boundary is "stitched". After crossing the boundary twice the solution point is located.
When the BF locates a boundary, two input variables are selected on their suitability for the roles of Follow and Cross.

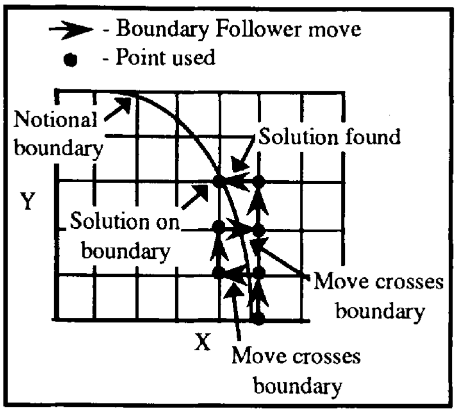


Figure 5.6 - Operation of the Boundary Follower

5.3.3 The Follow Role

Modifying the input variable allocated the *Follow role* , the *Follow variable*, will ideally, produce a point parallel to the boundary being followed. This however, may rarely be the case since it requires the boundary to be aligned with the Follow variable's axis. A *Follow move* modifies only the Follow variable of a point. All other input variable values are held constant. It is not necessary for a Follow move to stay on one side of the boundary. A Follow move may cross the boundary. Figure 5.7 shows a Follow move that stays on one side of a boundary and another that crosses it. The point a Follow move is made from is termed the *Follow move start point.* The point a Follow move is made to is termed the *Follow move end point.*

5.3.4 The Cross Role

Modifying the input variable allocated the *Cross role* , the *Cross variable*, must, after a few iterations, cross the boundary being followed. A *Cross move* modifies only the Cross variable of the Follow move end point. All other input variable values are held constant. It may be necessary to make a number of Cross moves before the boundary is crossed. If a Cross move does not cross the boundary then the Cross variable adjustment is increased using the Follow move end point value as a base. If this adjustment exceeds a threshold then a correctional phase is used and is discussed in section 5.3.7. Figure 5.7 shows Cross moves that cross the boundary with a step size of one and two.

The point a Cross move is made from (generate point) is termed the *Cross move start point.* The point a Cross move is made to (execute point) is termed the *Cross move end point*, which is normally the Follow move end point. A Cross move is termed a *Successful Cross move* if the start point and end point are on different sides of the boundary being followed. An *Unsuccessful Cross move* has start and end points on the same side of the boundary. The *Cross step size* stores the value the Cross variable is adjusted by, and is the difference between the Cross move start point and the Cross move end point. The *Maximum Cross step size* stores the maximum value the Cross step size can be.

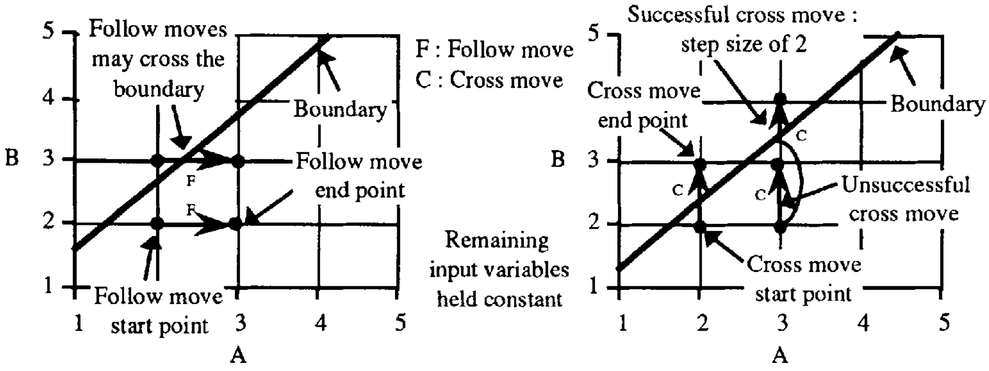


Figure 5.7 - Partial input planes showing Follow and Cross moves, with the Follow role allocated to variable A and Cross to B

5.3.5 Following a Boundary Using Follow and Cross Moves

To follow a boundary there is a cycle between Follow moves and Cross moves. First a Follow move is made. Second, one or more Cross moves are made until either the boundary is crossed or the maximum Cross step size is reached. On crossing the boundary, the cycle is closed and a further Follow move is made. If the maximum Cross step size is reached and the boundary has not been crossed then reorientation takes place. This involves a special search to relocate the boundary and preparation for the BF to continue (section 5.3.7). When following a boundary only the Follow variable or the Cross variable is modified at a time. The remaining input variables are held constant.

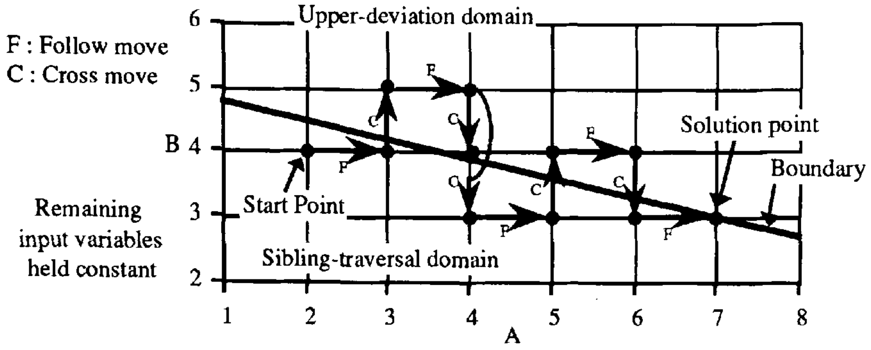


Figure 5.8 - A domain boundary being followed

To illustrate the process of boundary following, let us consider an untraversed node in a test procedure. This node poses the notional boundary located point problem. Figure 5.8 shows an arbitrary, notional boundary dividing domains causing upper-deviation and sibling-traversal. The solution is at point (7, 3). Input variable A has been allocated the Follow role and B, the Cross role. The point (2, 4) is selected as the Follow move start point.

A Follow move increases the Follow variable by 1. A Cross move increases the Cross variable by 1 from the Follow move end point. This Cross move crosses the boundary, hence a second Follow move can be made, from the Cross move end point. The Cross move, from (4, 5) to (4, 4), does not cross the boundary, so the Cross step size is increased by 1 to 2. After a few more moves the solution point is located by a follow move to point (7, 3).

5.3.6 Establishing the Correct Cross Move Direction

The Cross variable modification direction is unknown after the first follow move, taking place either when a new node is considered or after reorientation. A *bidirectional Cross search* looks in both directions for the boundary, by first making an increase move, then decrease, then increasing the Cross step size by one and making an increase move and so on. Using a bidirectional cross search after the first Follow move helps to ensure that the boundary is crossed for the second Follow move.

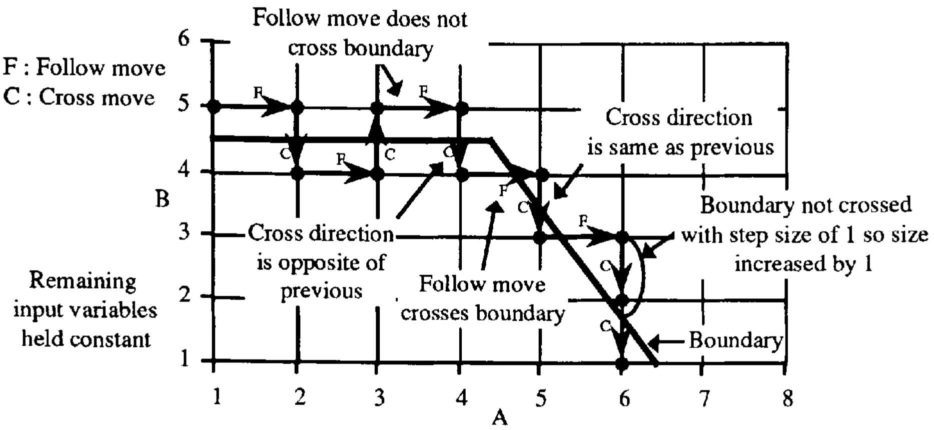

F : Follow move
C : Cross move
Follow move does not cross boundary
Cross direction is same as previous
Boundary not crossed with step size of 1 so size increased by 1
Cross direction is opposite of previous
Follow move crosses boundary
Boundary
Remaining input variables held constant
B
A
1
2
3
4
5
6
7
8
F
C

Figure 5.9 - Use of the Cross rule to determine the Cross direction

After the second Follow move a *unidirectional Cross search* is made, which uses a single direction to cross the boundary and is determined by the Cross rule. The *Cross rule* states that if the preceding Follow move does not cross the boundary then the new Cross direction is the opposite direction to the previous successful Cross move's. However, if the previous Follow move does cross the boundary then the new Cross direction is the same direction as the previous successful Cross move's. The Follow move start point's and end point's traversal-effects are used to determine if the Follow move has crossed the boundary. Figure 5.9 illustrates the Cross rule in action.

5.3.7 Reorienting the Boundary Follower

When the maximum Cross step size does not enable the boundary being followed to be crossed then, for the BF to continue, the boundary must be relocated and changes made to the role allocations. This process is called reorientation. Reasons for the BF being unable to cross the boundary are :

- another border has been encountered
- the nonlinear boundary being followed rapidly moved away from its previous "course"
- the maximum cross step size is not large enough

Figure 5.10 shows a partial input plane where two notional borders intersect. The BF has been following the "vertical" border in a "northerly" direction, when the "horizontal" intersecting border is encountered. Using Cross variable A, the BF is unable to cross the boundary with the maximum Cross step size of three.

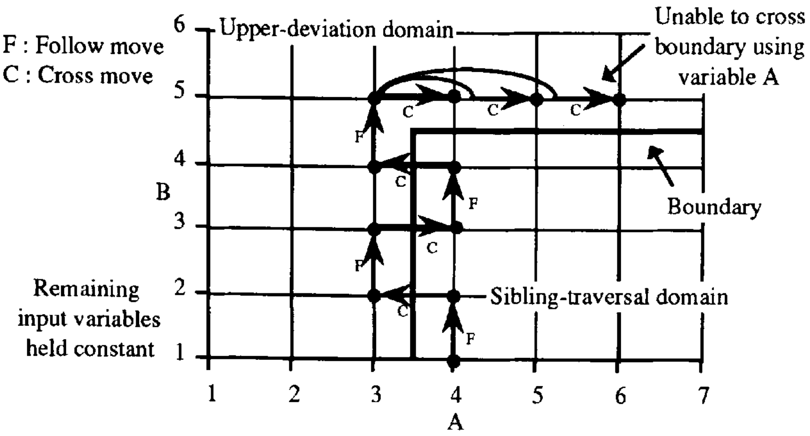


Figure 5.10 - A border change which renders the BF unable to cross the boundary with Cross variable A, and Follow variable B

Reorientation requires the boundary to be crossed from the preceding Follow move's end point. This is achieved by a search pair along a central line. Figure 5.11 illustrates each of these.

The *central line* is in the same axis as the Follow variable and runs through the preceding Follow move's end point, which is termed the *base point.* The *search pair* consist of an increase and a decrease point. The first search pair move from the base point by a step of size one in the reverse Follow direction and steps of size one in both directions for the Cross variable. If the boundary is not crossed by either of the search pair points then further search pairs are made further from the base point along the central line, by increasing the step in the reverse Follow direction by one and using the same Cross variable values as the first search pair.

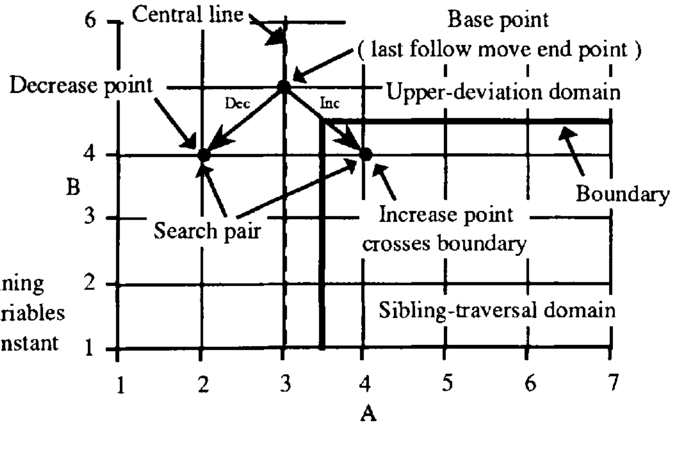


Figure 5.11 - Reorienting the BF

After execution of a search pair, their traversal-effects are analysed to determine if the boundary has been crossed. If so, reorientation can be completed and boundary following can recommence. To complete, the allocation of roles to input variables are swapped and a Follow direction is determined. If only one search pair point crossed the boundary then the Cross variable direction used is the new Follow direction. However, if both search pair points crossed the boundary then the Follow direction remains unchanged. To help ensure the boundary is crossed after the first Follow move the Cross search becomes bidirectional.

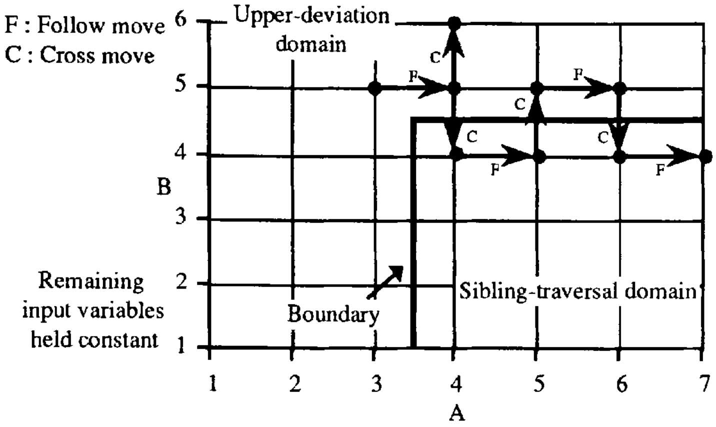


Figure 5.12 - Continuation of boundary following subsequent to reorientation with Follow variable A, Cross variable B and Follow direction of increase

Figure 5.11 shows that the search pair's increase point crossed the boundary from upper-deviations to sibling-traversals. Thus the new Follow direction is increase and variable A is allocated the Follow role and B, Cross. Figure 5.12 shows the BF continuing along the newly encountered border.

5.3.8 Locating a Point to Commence Boundary Following From

Before boundary following can commence a point that causes sibling-traversal and is adjacent to the boundary to be followed, must be located. This point, termed the *Central point*, determines which border will be followed when there is more than one border. A modification of the LP heuristic is used for this purpose since it has shown to be effective and efficient at locating points adjacent to a boundary.

5.3.9 Initial Allocation of the Follow and Cross Roles

When the Central point has been located, the Follow and Cross roles must be allocated to input variables. Selecting which input variables to use, involves increasing and decreasing each of the input variables by 1 from the Central point. The number of points produced is equal to twice the number of input variables. Each point's traversal-effect is stored.

The two traversal-effects for each input variable are considered in turn from the first input variable to the last (the same order the input variables have just been modified in). The increase and decrease traversal-effects for each input variable are compared with the traversal-effect of the Central point to determine a suitable role. If modifications to an input variable cause no change in the predicate value to the Central point's predicate value (input variable does not have considered-predicate influence) then the variable is not suitable for either role. Input variables that are not influential in the considered predicate are avoided since the border segment adjacent to the Central point does not exist for that input variable. Otherwise suitability is determined according to table 5.6, if the Central point has a predicate value greater than the boundary's predicate value (normally 0) and table 5.7 if the Central point has a predicate value less than the boundaries.

		Inc point trav-eff		
		+ST	-ST	UD
Dec point trav-eff	+ST	F	C	C
	-ST	C	F	
	UD	C		

Table 5.6 - Follow and Cross role suitability when the Central point's predicate value is greater than the boundary's predicate value. F - Follow; C - Cross.

		Inc point trav-eff		
		+ST	-ST	UD
Dec point trav-eff	+ST	F	C	
	-ST	C	F	C
	UD		C	

Table 5.7 - Follow and Cross role suitability when the Central point's predicate value is less than the boundary's predicate value. F - Follow; C - Cross.

If an input variable's traversal-effects are the same, this indicates that both points were on the same side of the boundary and that the input variable suits the Follow role. However, when there is two upper-deviations the input variable does not suit the Follow role, since it is not considered predicate influential. The Follow allocation is avoided since the upper-deviation domain may have other boundaries that are adjacent to the boundary located and these may interfere with the BF. If an input variable's traversal-effects are different, indicating the boundary was crossed, then the variable suits the Cross role.

Roles are allocated on a first suitable, first allocated basis. Once the allocation of a role to an input variable has taken place the role is not reallocated. Suspicion that the Follow role may be unallocated led to holding an input variable in reserve for the Follow role. After the Cross allocation has been made, should a further input variable suit the Cross role, it is held as the reserve for the Follow role. Should the Follow role be unallocated after all input variables have been considered then the variable held in reserve is allocated the Follow role. It is necessary for both roles to be allocated for the BF to follow a boundary, otherwise the BF terminates.

5.3.10 Phases of the Boundary Follower Heuristic

The BF has four phases. The first, *Obtain-a-close-point*, (OCP) locates a point adjacent to the boundary to be followed, in preparation for the next phase. The second phase, *Determine-initial-follow-and-cross-details*, (DIFCD) selects two input variables to allocate the Follow and Cross roles, in preparation for the next phase. These two phases are used once, directly after the BF is applied to a node.
The third phase, *Follow-boundary*, (FB) does so using the current role allocations.
The fourth phase, *Reorient-boundary-follower*, (RBF) relocates the lost boundary, swaps the roles and determines a new Follow direction for return to the FB phase.

5.3.11 Components and Functions of the Boundary Follower

The BF's pseudo code is in Appendix A3.4.

5.3.11.1 First Iteration Set-up, Generator and Evaluator

The first iteration set-up component specifies the current phase as Obtain-a-close-point and sub-phase as Determine-linearity. The OCP base point is set to a sibling-traversal point with the closest predicate value to the expected boundary. The first input variable shall be considered and increased by 1.
The generator component applies the input point generation instructions to the generate point producing the execute point that the test procedure will execute on.
The evaluator component is called after the test procedure has executed on the execute point. If the BF has been successful, control returns to the HATS-harness, otherwise evaluation is performed dependent upon the BF's phase and, with the Obtain-a-close-point phase, its subphase.

5.3.11.2 Obtain-a-close-point Phase

This phase locates a point that causes sibling-traversal and is adjacent to a boundary. This phase is a modification of the LP (section 5.2), and consists of three subphases; Determine-linearity (DL), Predictor and Creeper.
Before any of the sub-phases are called, the BF evaluator checks if a boundary has been crossed. If so, the closest point found is already adjacent to a boundary in at least one dimension (input variable), hence the considered input variable is abandoned.

The Predictor sub-phase evaluation is used after the execution of the Predicted point and prepares for the Creeper subphase.

The Creeper subphase is used if the considered predicate is deemed to be nonlinear or after the Predictor phase. The Creeper makes small steps with the considered input variable in the specified direction if progress is being made toward the expected boundary. Otherwise, the considered input variable is abandoned.

If the considered input variable is abandoned then a check is made to see if all the input variables have been considered and, if so, prepares for the Determine-initial-follow-and-cross-details phase, where the closest point found becomes the Central point. Alternatively, the Determine-linearity subphase is prepared for on the next input variable. Unlike the LP, there are no termination criteria based on unpromising effects in the Obtain-a-close-point phase.

5.3.11.3 Determine-initial-follow-and-cross-details Phase

This phase produces and analyses points surrounding the Central point to allocate the Follow and Cross roles, in preparation for the Follow-boundary phase. In the LP's terminator description (section 5.2.2.6), we saw how the LP was applied to node 6 of the QUADRATIC from the point (70, 50, 9), without success. Over the remainder of this section, examples are presented on how the BF solves this problem. Figure 5.13 shows the modifications made to the input variables from the Central Point (70, 50, 9) by the Determine-initial-follow-and-cross-details phase. Each input variable is increased and decreased by 1 to obtain the traversal-effects necessary to allocate the roles.

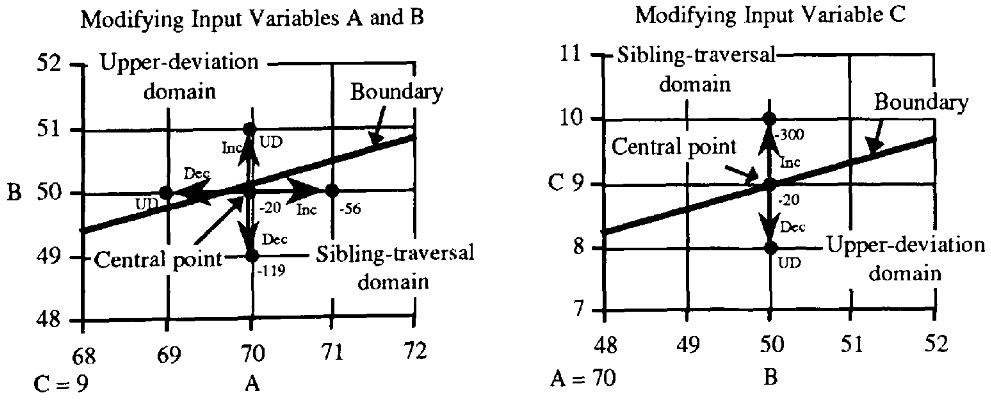


Figure 5.13 - Modifications made to the QUADRATIC's input variables during the Determine-initial-follow-and-cross-details phase of the BF

Table 5.8 shows each input variables' traversal-effects, the role each variable suits defined by table 5.7 (Central point < boundary predicate value) and the preliminary and final role allocation.

Var	Traversal-effect		Role	Role allocation	
	Decrease	Increase	suited	Preliminary	Final
A	UD	-ST	Cross	Cross	Cross
B	-ST	UD	Cross	Res Follow	Follow
C	UD	-ST	Cross		

Table 5.8 - Progress to the final allocation of roles in the Determine-initial-follow-and-cross-details phase with the QUADRATIC

Interestingly none of the input variables' traversal-effects immediately suit the Follow role. However, input variable B is held in reserve for Follow and is allocated the role finally. To complete preparation for the Follow-boundary phase, the Follow direction is set to increase and the Central point will be the first Follow move start point.

5.3.11.4 Follow-boundary Phase

This phase follows a boundary adjacent to the Central point or the Reorient-boundary-follower's base point. Continuing from the Obtain-a-close-point phase example (section 5.3.11.3), which specified that input variable A is Cross, B is Follow, Follow direction is increase, Cross search is bidirectional and a Follow move is to be made first. The Follow-boundary phase starts from the Central point (70, 50, 9), which causes a negative sibling-traversal and has a predicate value of -20. Table 5.9 and figure 5.14 show this phases operation.

First a Follow move is made from the Central point, by increasing the Follow variable, B, by 1. Since it is the first Follow move a bidirectional Cross search is used. The first Cross move increases the Cross variable, A, by 1 from the Follow move end point. This move does not cross the boundary, so a decrease move is made from the Follow move end point. This again does not cross the boundary, so the Cross step size is increased by 1 to 2, and the next increase move made. This move does not cross the boundary so a decrease move is made, then the Cross step size is increased again and an increase move made. This move does cross the boundary, so a Follow move can be made from the Cross move end point.

FB	Var	Phase	Generate point			Execute point			Trav	Pred	Next
Iter			A	B	C	A	B	C	effect	value	action
1	B	FB F	70	50	9	70	51	9	UD		C +1
2	A	FB C	70	51	9	71	51	9	UD		C -1
3	A	FB C	70	51	9	69	51	9	UD		C +2
4	A	FB C	70	51	9	72	51	9	UD		C -2
5	A	FB C	70	51	9	68	51	9	UD		C +3
6	A	FB C	70	51	9	73	51	9	-ST	-27	F
7	B	FB F	73	51	9	73	52	9	UD		C +1
8	A	FB C	73	52	9	74	52	9	UD		C +2
9	A	FB C	73	52	9	75	52	9	UD		C +3
10	A	FB C	73	52	9	76	52	9	-ST	-32	F
11	B	FB F	76	52	9	76	53	9	UD		C +1
12	A	FB C	76	53	9	77	53	9	UD		C +2
13	A	FB C	76	53	9	78	53	9	UD		C +3
14	A	FB C	76	53	9	79	53	9	-ST	-35	F
15	B	FB F	79	53	9	79	54	9	UD		C +1
16	A	FB C	79	54	9	80	54	9	UD		C +2
17	A	FB C	79	54	9	81	54	9	NT	0	SUCC

Table 5.9 - HATS run excerpt : BF in the Follow-boundary phase considering node 6 of the QUADRATIC and locating a solution point

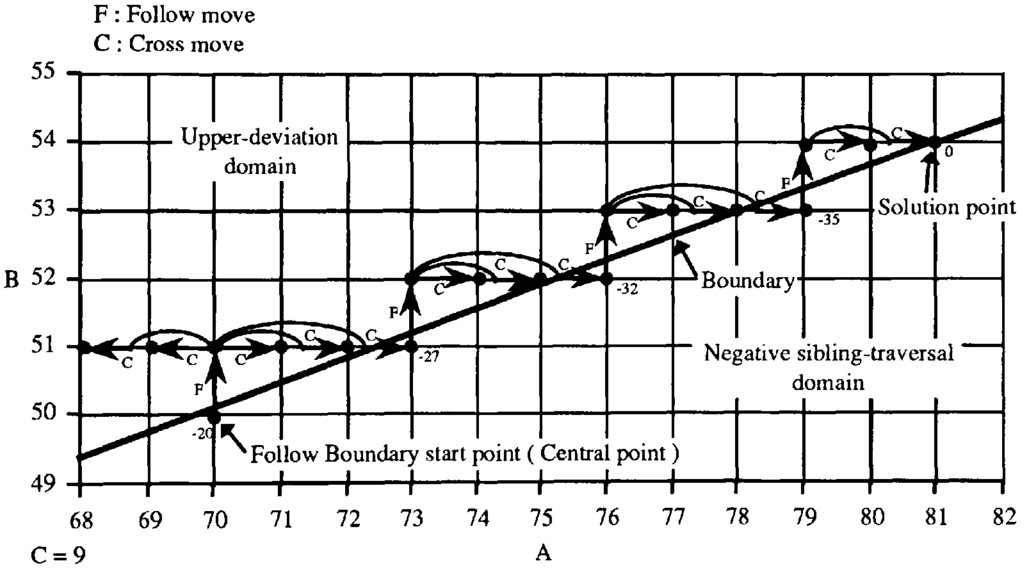


Figure 5.14 - (A, B) partial input plane showing Follow-boundary phase moves to a solution point for node 6 of the QUADRATIC

The next Cross move is unidirectional since the Cross rule has determined that the direction of increase shall be used. From here on the BF follows the boundary and locates a solution (81, 54, 9) after 17 iterations and produces only boundary located points.

The Cross moves after the first Follow move give a "mushroom" appearance, however, in subsequent Follow moves, there is only half a "mushroom", where the Cross rule has been applied. From the second Follow move onwards a distinct "pattern" of moves is established, which appears in this example as a Follow move then Cross moves' half "mushroom".

It is interesting to consider what the BF's behaviour would be if the allocation of roles in this example were swapped. Figure 5.15 shows the same notional boundary followed to the same solution with role allocations of A, Follow and B, Cross. With the first role allocation (figure 5.14) there are 13 upper-deviations, three negative sibling-traversals and a total of 17 iterations in the trace to the solution point. With the second role allocation (figure 5.15) there are 10 upper-deviations, 12 negative sibling-traversals and a total of 23 iterations in the trace to the solution point. In the first allocation, upper-deviation points are predominant. Whereas in the second allocation, there is more or less an even balance of points on both sides of the boundary. The notional boundary is better covered, but at the expense of a further six iterations.

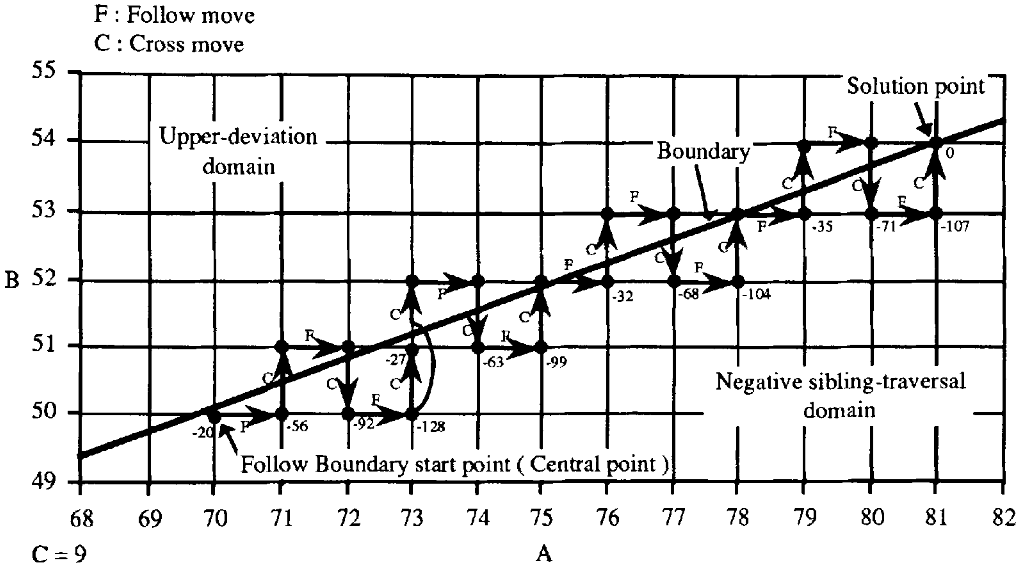


Figure 5.15 - Partial input plane showing Follow-boundary phase moves, had variable A been allocated Follow and B allocated Cross, to a solution point for node 6 of the QUADRATIC

5.3.11.5 Reorient-boundary-follower Phase

When the Follow-boundary phase cannot cross the boundary this phase relocates the boundary and prepares the Follow-boundary phase for continuation. The operation of this phase is described in section 5.3.7.

5.3.12 Known Limitations

The BF described has a number of limitations :

- If an input domain has more than one border segment, the choice of border segment to follow is determined by the point the OCP phase selects to start with.

- There are no heuristic termination criteria in the Follow-boundary or Reorient-boundary-follower phases.
- Once the two roles have been allocated, no other input variables are considered whilst on the considered node.
- The Follow direction is fixed on increase the first time the Follow-boundary phase is used.
- The orientation of the boundary with respect to the input space axes is not taken into consideration when the roles are allocated.
- A domain's boundary is followed in two dimensions, however a boundary may exist in many dimensions.

5.4 Installing the LP and BF into the HATS Harness

The LP and BF must be located in the heuristic selection order. The new order is shown in figure 5.16

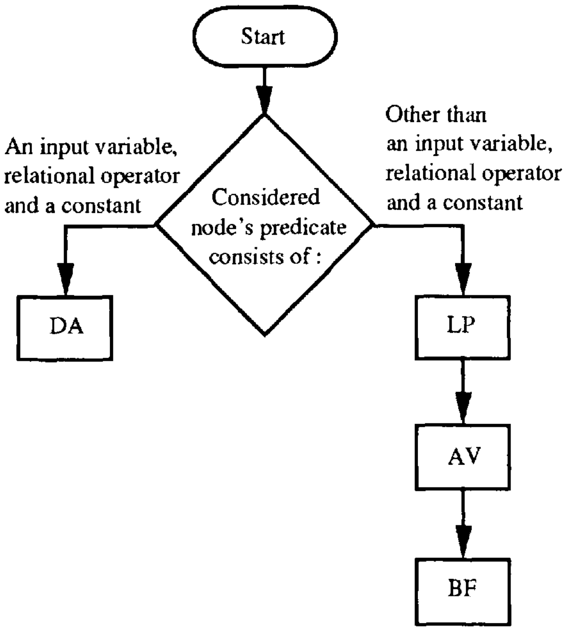

Start
An input variable, relational operator and a constant
Considered node's predicate consists of :
Other than an input variable, relational operator and a constant
DA
LP
AV
BF

Figure 5.16 - New heuristic selection order with the LP and BF heuristics added

5.5 The Second Round of Mutation Analysis

5.5.1 Experimental Set-up

The unrevealed mutants from the first round are the subjects of this second round. The same initial points are used as in the first round. However, the node iteration threshold has increased, tenfold, to 500, as it is anticipated that the BF may require more than 50 iterations.

Of the four heuristics only the DA and BF are used. The AV is not used since its termination criteria require improvement to prevent it from continuing needlessly. The LP is not used since it is virtually the same as the BF's Obtain-a-close-point phase. To aid identification, 100 has been added to the mutation identifier of the first round. Hence mutant QM01 becomes QM101.

5.5.2 Statement Analysis Results

The five unrevealed first round, statement analysis mutants are the subjects of this second round. The BF located points that revealed four of the five mutants and are shown in table 5.10. Table 5.11 has the unrevealed mutant.

Mutant	Initial point			No iters	Central point			Revealing point		
	A	B	C	on node	A	B	C	A	B	C
QM106	561	749	414	171	339	749	414	414	828	414
QM113	801	319	363	35	71	321	363	75	330	363
QM147	337	907	168	400	1225	907	168	1512	1008	168
QM149	322	517	102	369	656	517	102	918	612	102
Number of mutants			4							

Table 5.10 - Second round statement analysis mutants revealed by the BF on node 6

Mutant	Initial point			No iters	Central point			Last point		
	A	B	C	on node	A	B	C	A	B	C
QM148	743	884	589	500	332	884	589	524	1111	589
Number of mutants			1							

Table 5.11 - Second round statement analysis mutants unrevealed by the BF on node 6 in 500 iterations

Table 5.10 shows that the amount the Central point is adjusted by to produce the mutant revealing point increases as the number of iterations the BF takes to locate the revealing point increases. If the spatial distance is derived then the comparison can easily be made. *Spatial distance* is the distance from point to point in n-dimensional space. For example, the spatial distance from the Central point to the revealing point for QM113 is (4, 9, 0) as only A and B have changed.

However, this can be deceiving as it does not take the actual boundary followed into consideration and assumes it to be a straight line. Nevertheless, in the above second round mutants the notional boundary in the area of input space explored is virtually a straight line.

With the single unrevealed mutant QM148 (table 5.11), the BF terminated after reaching the node iteration threshold without locating a solution. The BF had located the notional boundary and followed it accurately, however there were no solutions along the part of the notional boundary that could be followed in 500 iterations. An exhaustive search of the input space, with C constant at 589, found the next solution on the notional boundary followed at (589, 1178, 589). Clearly, the BF had not missed any solutions and given a higher node iteration threshold, should locate this solution. It is estimated that a further 140 to 160 iterations would be needed by the BF to locate this solution .

5.5.3 Predicate Analysis Results

The six unrevealed first round predicate analysis mutants are the subjects of this second round. Mutant QM26 was not considered since the modifications to the DA necessary to reveal the mutant are made later (section 5.7). Two of the five mutants were revealed by the BF (table 5.12). However, the remaining three mutants were not revealed (table 5.13) by the BF before the node iteration threshold was met. Nevertheless, the boundary was followed accurately and no solutions were overlooked.

Mutant	Initial point			No iters	Central point			Revealing point		
	A	B	C	on node	A	B	C	A	B	C
QM116	421	821	334	436	505	821	334	750	1001	334
QM117	104	814	289	15	574	814	289	576	816	289
Number of mutants			2							

Table 5.12 - Second round predicate analysis mutants revealed by the BF on node 6

Mutant	Initial point			Central point			Last point			Next solution		
	A	B	C	A	B	C	A	B	C	A	B	C
QM118	252	469	652	95	497	651	190	703	651	651	1302	651
QM119	654	109	554	6	115	552	53	342	552	138	552	552
QM133	427	188	608	15	190	602	73	420	602	602	1204	602
Number of mutants			3									

Table 5.13 - Second round predicate analysis mutants unrevealed by BF on node 6 in 500 iterations

An exhaustive search of the input space located where the next solutions are on the notional boundary followed (table 5.13). Thus, had the node iteration threshold been higher, the BF should reveal all five mutants.

5.5.4 Improving Upon the AV

When considering a linear node the LP should take fewer iterations to locate a solution than the AV. A comparison is made between the AV and BF's Obtain-a-close-point phase, which is the same as the LP, with the exception that it does not have the LP's termination criteria.

Mutant	AV iters	BF's OCP iters
QM6/106	9	5
QM13/113	11	4
QM16/116	7	4
QM17/117	9	5
QM18/118	9	4
QM19/119	11	4
QM33/133	10	4
QM47/147	9	5
QM48/148	10	5
QM49/149	9	4
Average	9.4	4.4

Table 5.14 - Iterations taken by the AV and BF on node 4 or 5 of the QUADRATIC

Table 5.14 shows the iterations taken by the AV in round one and by the BF in round two, considering either node 4 or 5 of the QUADRATIC. The BF consistently took less iterations than the AV. On average the BF took approximately half the iterations of the AV.

5.6 The Third Round of Mutation Analysis

5.6.1 Experimental Set-up

To reveal the four remaining mutants we are faced with a choice, either to increase the node iteration threshold or to try a different initial point. The BF's ability to accurately follow a boundary has been established, so the alternative is considered. In this round new initial points are generated, in the range 0 to 9 for input variables A, B and C (table 5.15). The node iteration threshold remains at 500 and only the DA and BF are used. To identify third round mutants, 100 has been added to the identifier used in the second round.

5.6.2 Results

All four mutants were revealed by the BF (table 5.15). The new initial points did not reveal any mutant. With mutants QM219 and QM233, a revealing point was located in the BF's Obtain-a-close-point phase. The remaining mutants were revealed while the BF was following a notional boundary.

Mutant	Initial point			No iters	Central point			Revealing point		
	A	B	C	on node	A	B	C	A	B	C
QM218	8	0	3	19	1	3	3	3	6	3
QM219	1	7	9	7				2	8	8
QM233	6	3	9	5				1	6	9
QM248	7	9	4	17	6	9	4	9	12	4
Number of mutants			4							

Table 5.15 - Third round statement analysis mutants revealed by the BF on node 6

5.7 DA Improvements

Mutant QM26 was not revealed because the DA did not generate a negative value for input variable A. The DA has been modified to generate three points rather than one. The considered input variable takes, for separate iterations, the value of the predicate constant minus 1, the constant, and the constant plus 1. The remaining input variables are constant. This is boundary value test data. Now the DA is not concerned about the considered predicate's relational operator and has thus simplified the heuristic. The improved DA revealed QM26 with the point (-1, 781, 920).

5.8 Mutation Analysis Discussion and Conclusions

The BF has demonstrated that it can accurately follow a boundary in two dimensions, in the QUADRATIC's input space and locate points that the AV is unable to. The static node iteration threshold of 500 limits the distance the BF can follow a boundary. If the threshold is reached before a solution then the HATS-harness will deem the node infeasible when there may be a solution lying on the boundary. However, the BF is unable to determine if a solution does lie on the boundary or do some analysis to determine if it is heading in a promising direction.

Once the BF is following a boundary, only boundary located points are generated. The BF's Obtain-a-close-point phase, which is virtually the same as the LP, uses less iterations than the AV and generates boundary located points. The remaining first round mutant has been revealed by an improved DA that generates boundary located points.

6 The Triangle Classification Problem

6.1 Introduction

Triangle classification programs have become a common "benchmark" for test data generators (Deason, et al, 1991; Inamura, 1989; DeMillo and Offutt, 1991; Ramamoorthy, et al, 1976; Duran and Ntafos, 1984). Unfortunately, there is considerable variation in the coding of the triangle classifiers, which make comparison difficult.

This chapter presents the branch testing of a nested Ada classification procedure and its components, by HATS and random test data generation. HATS uses the DA, LP and BF heuristics. The AV is not included since its termination criteria require improvement.

The triangle classification procedures present different problems to the QUADRATIC. These include a significantly increased number of paths (and their corresponding domains), nested procedures and different path functions.

6.2 Experimental Conditions

The main procedure, TRIANGLE_COMPLETE, consists of three procedures; TRIANGLE, TRIANGLE_2 and RIGHT_ANGLE_CHECK. These four procedures are the subjects of HATS. Each procedure's function is described later in this chapter.

Each procedure has three integer input variables A, B and C which are the lengths of each side of a triangle. A single, hand selected point, in the range ±100 for each of the three input variables, is used to start each HATS harness run. Should the HATS harness be consistently unable to find a branch's solution point, hand selection would enable traversal of the branch and consideration of the branches deemed infeasible. The node iteration threshold is 500 and the LP and BF consider input variables in the order A first, B second and C third.

6.3 TRIANGLE Experiments

The TRIANGLE procedure (figure 6.1) checks that the given side lengths are of a legal triangle. Lines 4 to 6 check that each side's length is greater than zero. If these checks are passed, lines 8 to 10 check that twice each side's length is less than the perimeter size. If all these checks are passed, the triangle is legal and the next procedure, TRIANGLE_2, is called. To facilitate unit testing this call (line 11) has been commented out.

```
Control flow tree node number
    Line number
          Ada statements

     1  procedure TRIANGLE
          ( A, B, C in INTEGER; TRI_KIND out TRI_TYPE ) is
     2  P : INTEGER;
     3  begin
 1   4    if ( A > 0 ) then
 3   5      if ( B > 0 ) then
 5   6        if ( C > 0 ) then
 7   7          P := ( A + B + C );
     8          if ( ( 2 * A ) < P ) then
 9   9            if ( ( 2 * B ) < P ) then
11  10              if ( ( 2 * C ) < P ) then
13  11--              TRIANGLE_2 ( A, B, C, TRI_KIND );
    12              else
12  13                TRI_KIND := NOT_A_TRIANGLE;
    14              end if;
10  15            end if;
 8  16          end if;
    17        else
 6  18          TRI_KIND := NOT_A_TRIANGLE;
    19        end if;
    20      else
 4  21        TRI_KIND := NOT_A_TRIANGLE;
    22      end if;
    23    else
 2  24      TRI_KIND := NOT_A_TRIANGLE;
    25    end if;
    26  end TRIANGLE;
```

Figure 6.1 - The TRIANGLE procedure

The DA and LP should satisfy branch adequacy without the BF being used, since the conditions in lines 4 to 6 involve only an input variable and a constant and the conditions in lines 8 to 10 have a linear relationship with the input variables.

The TRIANGLE procedure's control flow tree (figure 6.2) has 12 branches and 13 nodes. The longest path consists of 7 nodes. Table 6.1 contains the initial points for the three HATS harness runs.

HATS	Input variable		
run	A	B	C
T1	-17	-14	-39
T2	57	24	43
T3	7	9	6

Table 6.1 - TRIANGLE procedure's initial points

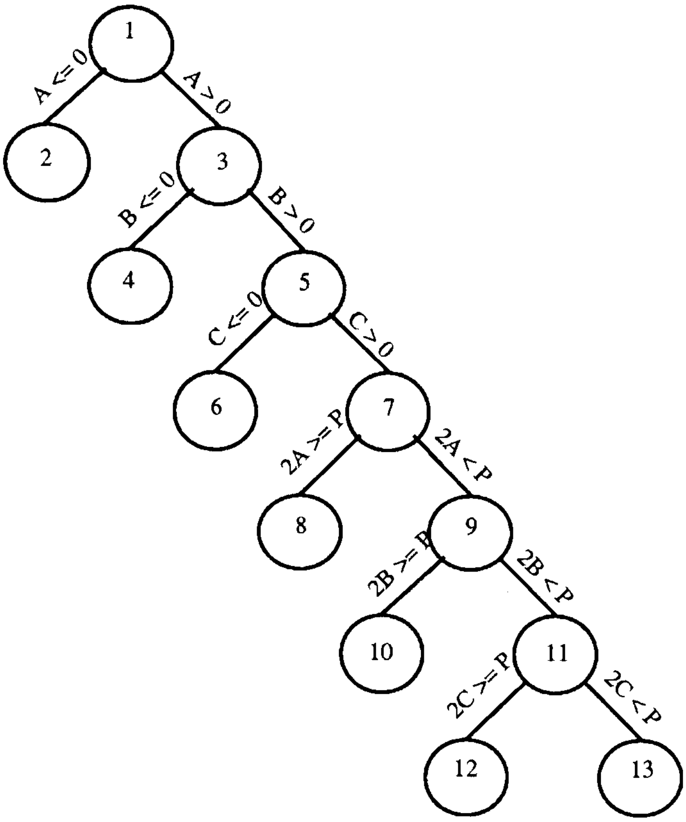


Figure 6.2 - Control flow tree of the TRIANGLE procedure

6.3.1 Run T1

Table 6.2 contains the traversal results for this run.

Initial point path		1,2	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	4
5	DA	3	6
7	DA	3	9, 11, 13
8	LP	1	
10	LP	2	
12	LP	3	

Table 6.2 - Run T1 traversal results

The DA generated boundary value test data for nodes 3, 5 and 7. The points for node 3 were : (-1, -14, -39), (0, -14, -39) and (1, -14, -39); node 5 (1, -1, -39), (1, 0, -39) and (1, 1, -39); node 7 (1, 1, -1), (1, 1, 0) and (1, 1, 1).

On nodes 8, 10 and 12 the LP chose the node start point (1, 1, 1) which was generated by the DA on node 7. On node 8, in the Determine-linearity phase, the LP increased variable A by 1 producing the point (2, 1, 1), which traversed node 8. The first iteration on node 10 produced an upper-deviation, (table 6.3) which caused the

LP to abandon input variable A. The second iteration modified variable B and produced a solution point.

Iter	Var	Generate point			Execute point			Trav	Pred	Next
		A	B	C	A	B	C	effect	value	action
1	A	1	1	1	2	1	1	UD		TNV
2	B	1	1	1	1	2	1	NT	0	SUCC

Table 6.3 - Run T1 excerpt : LP considering node 10 in the Determine-linearity phase

The first two iterations on node 12 (table 6.4) produced upper-deviations, taking control to nodes 8 and 10. The third iteration located a solution point modifying variable C.

Iter	Var	Generate point			Execute point			Trav	Pred	Next
		A	B	C	A	B	C	effect	value	action
1	A	1	1	1	2	1	1	UD		TNV
2	B	1	1	1	1	2	1	UD		TNV
3	C	1	1	1	1	1	2	NT	0	SUCC

Table 6.4 - Run T1 excerpt : LP considering node 12 in the Determine-linearity phase

This run demonstrates that if the node start point is immediately adjacent to a solution point then a simple search, where each input variable is increased and decreased, should find the solution. The LP's Determine-linearity phase can do this.

6.3.2 Run T2

Table 6.5 contains the traversal results for this run.

Initial point path		1, 3, 5, 7, 9, 11, 13	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	3	12
4	DA	3	8
6	DA	3	
10	LP	8	

Table 6.5 - Run T2 traversal results

On node 10 the LP chose the node start point (1, 24, 43) which is a product of the DA on the initial point. On iteration 2 (table 6.6), the LP causes an upper-deviation modifying variable A, and so considers the next variable.

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
1	1	24	43	2	24	43	ST	-21	-A
2	1	24	43	0	24	43	UD		TNV

Table 6.6 - Run T2 excerpt : LP considering node 10 in the Determine-linearity phase modifying variable A

Using variable B is much more successful as the linearity of node 10's predicate is identified following iterations 3 to 6 (table 6.7). Variable B is increased by 1 then decreased by 1, then increased by 2 and 3. The predicate values show there is a corresponding linear change, indicating that node 10's predicate is linear with respect to the input variable modified, B.

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
3	1	24	43	1	25	43	ST	-19	-B
4	1	24	43	1	23	43	ST	-21	+B
5	1	24	43	1	26	43	ST	-18	+B
6	1	24	43	1	27	43	ST	-17	PP

Table 6.7 - Run T2 excerpt : LP considering node 10 in the Determine-linearity phase identifying variable B as linear

A point close to the expected boundary that should cause sibling-traversal is predicted and used in iteration 7 (table 6.8). LP now modifies the predicted point in iteration 8, crossing an input domain boundary, to a solution point.

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
7	1	44	43	1	43	43	ST	-1	PP
8	1	43	43	1	44	43	NT	0	SUCC

Table 6.8 - Run T2 excerpt : LP considering node 10 in the Predicted-point phase, causing just sibling traversal then considered node traversal, modifying variable B

If you consider tables 6.6 to 6.8 and figure 6.3, you can see how the LP operated next to a boundary defined by upper-deviations, progressing to a solution point that has a predicate value of 0. The consideration of node 10 demonstrates a successful non-problematic application of the LP.

	45	*	*	1	0
	44	*	*	0	-1
	43	*	*	-1	-2
	42	*	*	-2	-3
	:	:	:	:	:
	28	*	*	-16	-17
B	27	*	*	-17	-18
	26	*	*	-18	-19
	25	*	*	-19	-20
	24	*	*	-20	-21
	23	*	*	-21	-22
	22	*	*	-22	-23
		-1	0	1	2
C=43				A	

Figure 6.3 - Run T2 (A, B) partial input plane : LP considering node 10

6.3.3 Run T3

Table 6.9 contains the traversal results for this run. Unexpectedly the LP failed on node 12, enabling the BF to find a solution point. For node 12 the LP chose a start point of (7, 9, 6) and made this the current base point. The Determine-linearity phase establishes that variable A is linear with respect to node 12 (table 6.10). Iteration 2 produces a point that is closer than the current base point. However, the base point is not updated during the DL phase as this would upset the phase.

Initial point path		1, 3, 5, 7, 9, 11, 13	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	3	10
4	DA	3	8
6	DA	3	
12	LP	16	
"	BF	14	

Table 6.9 - Run T3 traversal results

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
BP				7	9	6	ST	-10	
1	7	9	6	8	9	6	ST	-11	-A
2	7	9	6	6	9	6	ST	-9	+A
3	7	9	6	9	9	6	ST	-12	+A
4	7	9	6	10	9	6	ST	-13	PP

Table 6.10 - Run T3 excerpt : LP considering node 12 in the Determine-linearity phase modifying variable A

With the linearity of variable A established, a point close to the expected boundary is predicted (table 6.11 : iteration 5 and figure 6.4). However, an upper-deviation is produced. In an attempt to locate the boundary, the LP progressively moves from the predicted point back to the current base point (table 6.11 : iterations 6 to 10).

Iteration 11 produces a sibling-traversal, not a solution, crossing the same boundary crossed by the predicted point (iteration 5). The LP now abandons variable A.

Iter	Generate point			Execute point			Trav	Pred	Next
	A	B	C	A	B	C	effect	value	action
5	-3	9	6	-2	9	6	UD		B BP
6	-2	9	6	-1	9	6	UD		B BP
7	-1	9	6	0	9	6	UD		B BP
8	0	9	6	1	9	6	UD		B BP
9	1	9	6	2	9	6	UD		B BP
10	2	9	6	3	9	6	UD		B BP
11	3	9	6	4	9	6	ST	-7	TNV

Table 6.11 - Run T3 excerpt : LP considering node 12 in the Predicted-point phase, modifying variable A to locate a point that causes sibling-traversal after predicting a point and producing an upper-deviation

	8	*	*	*	*	*	*	-3	-4	-5	-6	-7	-8	-9
C	7	*	*	*	*	*	*	*	-5	-6	-7	-8	-9	-10
	6	*	*	*	*	*	*	*	*	-7	-8	-9	-10	-11
	5	*	*	*	*	*	*	*	*	*	-9	-10	-11	-12
		-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
B = 9								A						

Figure 6.4 - Run T3 (A, C) partial input plane : LP considering linear node 12 and failing

On setting up for variable B, the LP updates the base point and closest point stored, as modifying variable A produced the point (4, 9, 6) which is closer than the previous base point (7, 9, 6). From this new base point the LP went on to consider variables B, C and A again (table 6.12). Once again, in iteration 13, a point is produced that is closer to the expected boundary than the current LP base point. Having no success on all three input variables, the LP terminated, allowing the BF to be used.

Iter	Var	Generate point			Execute point			Trav	Pred	Next
		A	B	C	A	B	C	effect	value	action
BP					4	9	6	ST	-7	
12	B	4	9	6	4	10	6	UD		TNV
13	C	4	9	6	4	9	7	ST	-6	-C
14	C	4	9	6	4	9	5	UD		TFV
15	A	4	9	6	5	9	6	ST	-8	-A
16	A	4	9	6	3	9	6	UD		TERM

Table 6.12 - Run T3 excerpt : LP considering node 12 in the Determine-linearity phase having no success

The BF chose the start point (4, 9, 7). The DL subphase of the OCP phase, which is virtually the same as the LP, established that variable A is linear with respect to node 13 (table 6.13 : iterations 17 to 20 and figure 6.4). The prediction (iteration 21) produced an upper-deviation, resulting in the BF abandoning variable A. On variable B (table 6.13 : iterations 22 to 24 and figure 6.5) an upper-deviation is produced again resulting in variable B being abandoned.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
BP						4	9	7	ST	-6	
17	A	O DL	4	9	7	5	9	7	ST	-7	-A
18	A	O DL	4	9	7	3	9	7	ST	-5	+A
19	A	O DL	4	9	7	6	9	7	ST	-8	+A
20	A	O DL	4	9	7	7	9	7	ST	-9	PP
21	A	O PP	-2	9	7	-1	9	7	UD		TNV
22	B	O DL	4	9	7	4	10	7	ST	-7	-B
23	B	O DL	4	9	7	4	8	7	ST	-5	+B
24	B	O DL	4	9	7	4	11	7	UD		TNV

Table 6.13 - Run T3 excerpt : BF considering node 12 in the Obtain-a-close-point phase, having no success

On variable C, the BF makes a prediction (table 6.14 : iterations 25 to 30 and figure 6.5) without producing upper-deviations.

To explain why the BF found a solution when the LP should have, stems back to iteration 13. On this iteration the LP's Determine-linearity phase located a point that was closer to the expected boundary than the base point at the time. Iteration 14 produced an upper-deviation which resulted in the LP abandoning variable C. Eventually, after considering other variables, the LP terminated. The BF found a solution using variable C from the point produced in iteration 13. Had the LP returned to variable C using point (4, 9, 7) as a base, it would have located the solution.

Iter	Phase	Generate point			Execute point			Trav	Pred	Next
		A	B	C	A	B	C	effect	value	action
BP					4	9	7	ST	-6	
25	O DL	4	9	7	4	9	8	ST	-5	-C
26	O DL	4	9	7	4	9	6	ST	-7	+C
27	O DL	4	9	7	4	9	9	ST	-4	+C
28	O DL	4	9	7	4	9	10	ST	-3	PP
29	O PP	4	9	13	4	9	12	ST	-1	PP
30	O PP	4	9	12	4	9	13	NT	0	SUCC

Table 6.14 - Run T3 excerpt : BF considering node 12 in the Obtain-a-close-point phase succeeding by modifying variable C

	14	2	1	0	-1	-2
	13	1	0	-1	-2	-3
	12	0	-1	-2	-3	-4
	11	-1	-2	-3	-4	-5
C	10	-2	-3	-4	-5	-6
	9	-3	-4	-5	-6	-7
	8	-4	-5	-6	-7	*
	7	-5	-6	-7	*	*
	6	-6	-7	*	*	*
		8	9	10	11	12
A = 4				B		

Figure 6.5 - Run T3 (B, C) partial input plane : BF considering node 12 and succeeding

6.3.4 Comparison of HATS with Random Testing on the TRIANGLE Procedure

The random testing range for each input variable is ± 100, which is the same as HATS's initial point selection range. Each random testing run continues till all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs T1 to T3.

Node	HATS iterations	Random iterations
1	IPT	1
2	3	2.002
3	3	1.966
4	3	4.12
5	3	4.004
6	3	8.178
7	3	8.012
8	1	48.22
9	IPT/CT	9.336
10	2-8	51.896
11	IPT/CT	11.278
12	3-30	45.74
13	IPT/CT	15.826

Key :

IPT	Initial point traversed
CT	Coincidentally traversed

Table 6.15 - The number of iterations taken by HATS and random testing for each of TRIANGLE's nodes

Table 6.15 shows that nodes 2 to 5 are comparable. On nodes 6 and 7, HATS took approximately half the iterations of random. While, the iterations for HATS on nodes 2 to 7 remained constant, on nodes 6 and 7 random took approximately twice the iterations of node 2 to 5. This is because the satisfying domain for nodes 6 and 7 is half the size of nodes 2 to 5. With nodes 8 to 13, HATS takes less iterations than random. Nodes 9, 11 and 13, which have a larger satisfying domain than nodes 8, 10 and 12, are all covered by HATS with the initial point or through coincidental traversal. The HATS solution to node 8 took only 1 iteration as HATS started on a point next to the solution found. Generally, HATS improves on random and at worst is equal to random.

6.4 TRIANGLE_2 Experiments

TRIANGLE_2 (figure 6.6) decides if the given side lengths represent an equilateral or isosceles triangle. If the lengths represent neither of these then normally RIGHT_ANGLE_CHECK would be called. However, to enable unit testing, this call has been commented out and TRIANGLE_2 is used independently of TRIANGLE.

which would normally invoke it. A value representing the triangle type is returned after identification.

```
Control flow tree node number
    Line number
        Ada statements

     1  procedure TRIANGLE_2
          ( A, B, C in INTEGER; TRI_KIND out TRI_TYPE ) is
     2  begin
1[   3    if ( A = B ) then
2[   4      if ( B = C ) then
4[   5        TRI_KIND := EQUILATERAL;
     6      else
5[   7        TRI_KIND := ISOSCELES;
     8      end if;
     9    else
3[  10      if ( A = C ) then
6[  11        TRI_KIND := ISOSCELES;
    12      else
7[  13        if ( B = C ) then
8[  14          TRI_KIND := ISOSCELES;
    15        else
9[  16--        RIGHT_ANGLE_CHECK ( A, B, C, TRI_KIND );
    17        end if;
    18      end if;
    19    end if;
    20  end TRIANGLE_2;
```

Figure 6.6 - The TRIANGLE_2 procedure

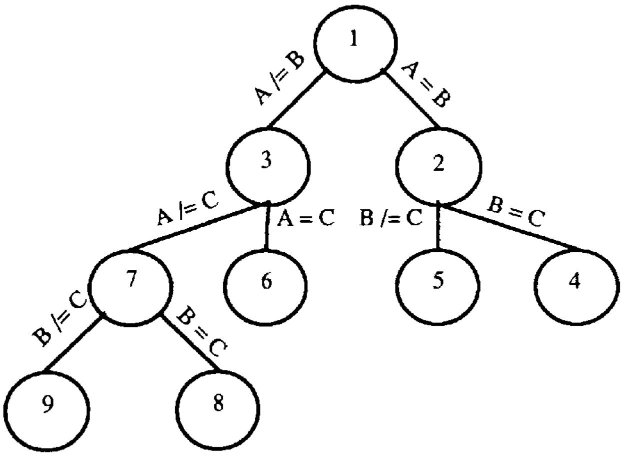


Figure 6.7 - Control flow tree of the TRIANGLE_2 procedure

TRIANGLE_2's control flow tree (figure 6.7) has 8 branches, 9 nodes and the longest path consists of 4 nodes. All conditions have a linear relationship with the input variables. Table 6.16 contains the initial points for three HATS harness runs.

HATS	Input variable		
run	A	B	C
T4	-27	-42	-77
T5	29	47	93
T6	1	4	3

Table 6.16 - TRIANGLE_2 procedure's initial points

6.4.1 Run T4

Table 6.17 contains the traversal results for this run.

Initial point path		1,3,7,9	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	6	5
6	LP	9	8
4	LP	8	

Table 6.17 - Run T4 traversal results

The LP considered nodes 2, 6, and 4 in that order. The DA is not suitable for any of TRIANGLE_2's conditions. On node 2, the LP used variable A to determine that the node is linear and predict a solution point. On node 6, the LP selected a base point that had been generated through node 2's consideration. Modifying variables A and B produced upper-deviations, however modifying variable C located a solution point. On node 4, the LP again selected a base point that had been generated through node 2's consideration. Again modifying variables A and B produced upper-deviations, but modifying variable C located a solution point.

This run presents a common obstacle for heuristics to manage. Heuristics attempt to generate points that are close to a domain boundary. When lower nodes in the control flow tree are considered a heuristic may choose one of these points close to a boundary higher in the control flow tree, as a base point. As the heuristic progresses the high nodes (interfering predicates) boundary may be crossed resulting in an upper-deviation. This may prevent any further progression with the current variable since there is no predicate value for the considered node's sibling, resulting in the current variable being abandoned.

The solution point for the lower node may not be located due to the upper-deviations affecting the active heuristic. The lower a node is in the control flow tree, the harder it is to locate a solution point, since there is a greater number of branches that can take control away from the considered node or its sibling, interrupting the active heuristic.

6.4.2 Run T5

Table 6.18 contains the traversal results for this run.

Initial point path		1, 3, 7, 9	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	6	5
6	LP	9	8
4	LP	8	

Table 6.18 - Run T5 traversal results

This run presents a similar situation to run T4, in that the LP selected start points for nodes 4 and 6 that are close to a boundary which, if crossed, will cause an upper-deviation. However, the LP is able to perform linearity determination and predict a successful point by modifying one of the variables without producing upper-deviations.

6.4.3 Run T6

Table 6.19 contains the traversal results for this run.

Initial point path		1, 3, 7, 9	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	4	6, 5
4	LP	3	
8	LP	6	

Table 6.19 - Run T6 traversal results

Again LP chose start points for nodes 4 and 8 that are close to an upper boundary and solution points for the considered nodes. All the solutions were found during the LP's Determine-linearity phase.

6.4.4 Comparison of HATS with Random Testing on the TRIANGLE_2 Procedure

The random testing range for each input variable is ± 100, which is the same as HATS's initial point selection range. Each random testing run continues till all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs T4 to T6.

Node	HATS iterations	Random iterations
1	IPT	1
2	4-6	192.534
3	IPT/CT	1.006
4	3-8	42652.4
5	IPT/CT	193.314
6	9	203.542
7	IPT/CT	1.01
8	6	196.224
9	IPT/CT	1.012

Key :	
IPT	Initial point traversed
CT	Coincidentally traversed

Table 6.20 - The number of iterations taken by HATS and random testing for each of TRIANGLE_2's nodes

Table 6.20 indicates that nodes 3, 7 and 9 have a large solution domain and consequently are easy to traverse. However, nodes 2, 4, 5, 6 and 8 are not as easy and random takes a higher number of iterations, especially on node 4. Nodes 2, 5, 6 and 8 have an equality predicate in the partial path to them. Node 4 has two equality predicates in the partial path to it. Each equality predicate in the partial path to a node reduces the dimensionality of the solution domain by one (White and Cohen, 1980). Consequently, random test data generation takes an increasing number of iterations for each equality predicate in the partial path to a considered node. HATS has taken significantly fewer iterations than random. There has not been any noticeable increase in iterations considering node 4 to nodes 2, 6 or 8, unlike random.

6.5 RIGHT_ANGLE_CHECK Experiments

This procedure (figure 6.8) checks the given side lengths to see if they represent a right angled scalene triangle or a non right angled scalene triangle. TRIANGLE_2 would normally call RIGHT_ANGLE_CHECK, however for unit testing RIGHT_ANGLE_CHECK is tested independently. All conditions have a non-linear relationship with the input variables.

```
Control flow tree node number
     Line number
          Ada statements

    1  procedure RIGHT_ANGLE_CHECK ( A, B, C in INTEGER;
    2                                   TRI_KIND out TRI_TYPE ) is
    3  begin
1   4   if ( ( ( A * A ) + ( B * B ) ) = ( C * C ) ) then
2   5    TRI_KIND := RIGHT_ANGLED_SCALENE;
    6   else
3   7    if ( ( ( B * B ) + ( C * C ) ) = ( A * A ) ) then
4   8     TRI_KIND := RIGHT_ANGLED_SCALENE;
    9    else
5  10     if ( ( ( A * A ) + ( C * C ) = ( B * B ) ) then
6  11      TRI_KIND := RIGHT_ANGLED_SCALENE;
   12     else
7  13      TRI_KIND := NON_RIGHT_ANGLED_SCALENE;
   14     end if;
   15    end if;
   16   end if;
   17  end RIGHT_ANGLE_CHECK;
```

Figure 6.8 - The RIGHT_ANGLE_CHECK procedure

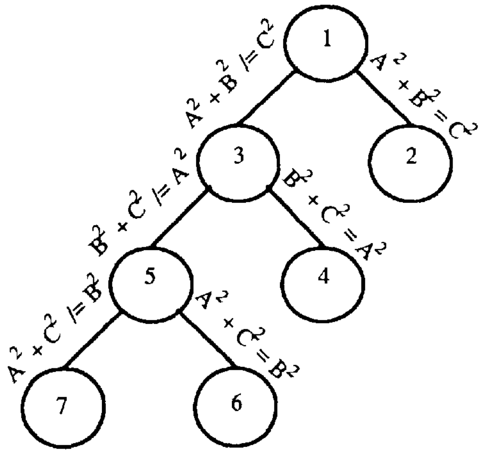


Figure 6.9 - Control flow tree of the RIGHT_ANGLE_CHECK procedure

The control flow tree (figure 6.9) has 6 branches, 7 nodes and the longest path consists of 4 nodes. Table 6.21 contains the initial points for three HATS harness runs.

HATS	Input variable		
run	A	B	C
T7	5	3	3
T8	10	20	30
T9	-27	-63	-97

Table 6.21 - RIGHT_ANGLE_CHECK procedure's initial points

6.5.1 Run T7

Table 6.22 contains the traversal results for this run.

Initial point path		1,3,5,7	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	8	
4	LP	6	
6	LP	12	
"	BF	17	

Table 6.22 - Run T7 traversal results

On node 2, the LP used variable A to determine that the node is non-linear (table 6.23 : iterations 1 to 4). Consequently the heuristic creeps to a solution point (table 6.23 : iterations 5 to 8).

Iter	Phase	Generate point			Execute point			Trav	Pred	Next	LP base point			
		A	B	C	A	B	C	effect	value	action	A	B	C	Val
1	DL	5	3	3	6	3	3	ST	36	-A	5	3	3	25
2	DL	5	3	3	4	3	3	ST	16	+A	5	3	3	25
3	DL	5	3	3	7	3	3	ST	49	+A	5	3	3	25
4	DL	5	3	3	8	3	3	ST	64	-A	5	3	3	25
5	CR	4	3	3	3	3	3	ST	9	-A	4	3	3	16
6	CR	3	3	3	2	3	3	ST	4	-A	3	3	3	9
7	CR	2	3	3	1	3	3	ST	1	-A	2	3	3	4
8	CR	1	3	3	0	3	3	NT	0	SUCC	1	3	3	1

Table 6.23 - Run T7 excerpt : LP creeping to a solution point, modifying variable A, on non-linear node 2

On node 4, the LP operated similarly to node 2. On node 6, the LP failed, but the BF succeeded. BF selected variable A for the Cross role and B for the Follow role. Figure 6.10 and table 6.24 illustrate the operation of the BF in the Follow-boundary phase, to the location of a solution point.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
24	B	FB F	1	3	3	1	4	3	-ST	-6	C 1
25	A	FB C	1	4	3	2	4	3	-ST	-3	C 1
26	A	FB C	1	4	3	0	4	3	-ST	-7	C 2
27	A	FB C	1	4	3	3	4	3	+ST	2	F
28	B	FB F	3	4	3	3	5	3	-ST	-7	C 1
29	A	FB C	3	5	3	4	5	3	NT	0	SUCC

Table 6.24 - Run T7 excerpt : BF finding a solution for non-linear node 6

	6	-26	-27	-26	-23	-18	-11	-2
	5	-15	-16	-15	-12	-7	0	9
B	4	-6	-7	-6	-3	2	9	*
	3	1	*	1	4	9	16	25
	2	6	5	6	9	14	21	30
		-1	0	1	2	3	4	5
C=3					A			

Figure 6.10 - Run T7 (A, B) partial input plane : BF considering node 6

Interestingly, the LP located solution points for the non-linear node 2 and 4. This occurred because the heuristic's start points are very close to the solution points located.

6.5.2 Run T8

Table 6.25 contains the traversal results for this run.

Initial point path		1,3,5,7	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	26	
"	BF	19	
4	LP	42	
"	BF	270	
6	LP	33	
"	BF	67	

Table 6.25 - Run T8 traversal results

The LP identified nodes 2, 4 and 6 as non-linear and crept to the closest point to the expected boundary it could locate for each node, then terminated. The BF took over and found solution points for each of these nodes.

The BF behaved as expected on nodes 2 and 4, but unusually on node 6, which deserves further investigation. Just after the initial allocation of the Follow and Cross roles the Reorient-boundary-follower phase was invoked three times. This appeared unusual since there are no "corners" (where two border segments meet) to navigate in this part of the sibling-traversal domain (figure 6.11).

	32	-115	-120	-123	-124	-123	-120	-115
	31	-52	-57	-60	-61	-60	-57	-52
B	30	9	4	1	*	1	4	9
	29	68	63	60	59	60	63	68
	28	125	120	117	116	117	120	125
	27	180	175	172	171	172	175	180
		-3	-2	-1	0	1	2	3
C=30					A			

Figure 6.11 - Run T8 (B, C) partial input plane : BF considering node 6 and reorienting three times

Figure 6.11 shows the boundary to be followed and an upper-deviation point (0, 30, 30). It is this upper-deviation point which caused the BF to behave unusually. In the BF's Determine-initial-follow-and-cross-details phase each input variable is increased then decreased by 1 from the central point (-1, 30, 30). Modifying variable A produced an upper-deviation and a positive-sibling-traversal, and modifying variable B produced a negative-sibling-traversal and a positive-sibling-traversal. Ideally variable B would be allocated Cross and variable A, Follow (figure 6.11). However, in the Determine-initial-follow-and-cross-details phase, variable A is modified before B, and produces two different traversal-effects. Therefore, variable A is allocated the Cross role and variable B the Follow role. With this allocation, modifying variable A will not locate a point on the other side of the notional boundary that may lead the BF to a solution point. The point causing upper-deviation has adversely affected the initial allocation of the Follow and Cross roles.

This is confirmed as the Follow-boundary phase is unable to cross the notional boundary (table 6.26 : iterations 51 to 54 and figure 6.11), as only negative-sibling-traversals were produced. Consequently the Reorient-boundary-follower phase is invoked and boundary crossing points located (table 6.26 : iterations 55 to 56), so that the roles could be reallocated. The Follow role is allocated to variable A and Cross to variable B. The Point (-1, 31, 30) is selected to start the Follow-boundary phase.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
51	B	FB F	-1	30	30	-1	31	30	-ST	-60	C 1
52	A	FB C	-1	31	30	0	31	30	-ST	-61	C 1
53	A	FB C	-1	31	30	-2	31	30	-ST	-57	C 2
54	A	FB C	-1	31	30	1	31	30	-ST	-60	OF+C
55	A, B	RBF	-1	31	30	0	30	30	UD		-C
56	A	RBF	-1	30	30	-2	30	30	+ST	4	FB

Table 6.26 - Run T8 excerpt : BF's first reorientation considering node 6

In the Follow-boundary phase, iteration 59 and 60 (table 6.27) crossed the notional boundary from the negative-sibling-traversal domain into an upper-deviation domain, then crossed the notional boundary again, coming out in the positive-sibling-traversal domain. Since the Follow move (iteration 60) crossed a boundary the Cross-rule determined that the Cross direction should be the same as the last direction to cross the notional boundary (decrease). Consequently, the BF could not cross back into the negative-sibling-traversal domain (iterations 61 and 62). This reorientation occurred as the Cross-rule had not taken the upper-deviation domain into consideration when it determined the Cross direction to use for iteration 61.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
57	A	FB F	-1	31	30	0	31	30	-ST	-61	C 1
58	B	FB C	0	31	30	0	32	30	-ST	-124	C 1
59	B	FB C	0	31	30	0	30	30	UD		F
60	A	FB F	0	30	30	1	30	30	+ST	1	C 1
61	B	FB C	1	30	30	1	29	30	+ST	60	C 2
62	B	FB C	1	30	30	1	28	30	+ST	117	OF+C
63	A, B	RBF	1	30	30	0	31	30	-ST	-61	-C
64	B	RBF	0	30	30	0	29	30	+ST	59	FB

Table 6.27 - Run T8 excerpt : BF's second reorientation considering node 6

The allocation of roles from the second reorientation rendered crossing the notional boundary very difficult (table 6.28 : iterations 66 and 67). A further reorientation took place, allocating the roles, Follow to variable A and Cross to B. With the best role allocation and the upper-deviation point out of the way the Follow-boundary phase continued (table 6.28 : iterations 70 to 73) without further interruptions and eventually located a solution point.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
65	B	FB F	1	30	30	1	31	30	-ST	-60	C 1
66	A	FB C	1	31	30	0	31	30	-ST	-61	C 2
67	A	FB C	1	31	30	-1	31	30	-ST	-60	OF+C
68	A, B	RBF	1	31	30	2	30	30	+ST	4	-Cr
69	A	RBF	1	30	30	0	30	30	UD		FB
70	A	FB F	1	31	30	2	31	30	-ST	-57	C 1
71	B	FB C	2	31	30	2	30	30	+ST	4	F
72	A	FB F	2	30	30	3	30	30	+ST	9	C 1
73	B	FB C	3	30	30	3	31	30	-ST	-52	F

Table 6.28 - Run T8 excerpt : BF's third reorientation considering node 6

In run T7, there are upper-deviation points dividing the notional boundary being followed (figure 6.10). Since the upper-deviation points were not used, the problem observed in this run did not occur. However, if they were used then there would be less of a problem as the notional boundary is on a diagonal, so it is possible to cross with both variables.

6.5.3 Run T9

Table 6.29 contains the node traversal results for this run.

Initial point path		1,3,5,7	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	LP	59	
"	BF	148	
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
4	LP	9	
"	BF	491	
Nodes unconsidered			
Considered node		Nodes unconsidered	
4		6	

Table 6.29 - Run T9 traversal results

The LP identified node 2 as non-linear and crept to the point closest to the expected boundary, then terminated. The BF found a solution without difficulty. The LP identified node 4 as non-linear, crept then terminated. The BF followed a boundary without difficulty till the node iteration threshold was reached, then terminated. Hence no solution was found. Node 6 was not considered as HATS terminated after considering node 4.

Node 4's consideration raises an important concern on choosing a value for the node iteration threshold. This was previously raised in section 5.5.3. In this case the area around the boundary followed and further along the boundary was searched for solutions, and none existed.

6.5.4 Comparison of HATS with Random Testing on the RIGHT_ANGLE_CHECK Procedure

The random testing range for each input variable is ± 100, which is the same as HATS's initial point selection range. Each random testing run continues till all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs T7 to T9.

Node	HATS iterations	Random iterations
1	IPT	1
2	8-207	5442.05
3	IPT/CT	1
4	6-312	6738.34
5	IPT/CT	1
6	29-100	9676.02
7	IPT/CT	1

Key :	
IPT	Initial point traversed
CT	Coincidentally traversed

Table 6.30 - The number of iterations taken by HATS and random testing for each of RIGHT_ANGLE_CHECK's nodes

Table 6.30 indicates that nodes 3, 5 and 7 have a large solution domain and consequently are easy to traverse. However, nodes 2, 4 and 6 are much harder, which can be seen by the increased number of iterations taken by both HATS and random testing compared to the iterations taken for TRIANGLE and TRIANGLE_2. HATS takes significantly fewer iterations than random testing. The large difference in HATS iteration range is due to the proximity of the initial point to the located solution point.

6.6 TRIANGLE_COMPLETE Experiments

The TRIANGLE_COMPLETE procedure (figure 6.12) is composed of the three procedures previously covered. The conditions in the procedure have a linear relationship with the input variables till the RIGHT_ANGLE_CHECK procedure. From there on, there is a non-linear relationship between the input variables and the conditions. The control flow tree (figure 6.13) has 26 branches, 27 nodes and the longest path consists of 13 nodes. Where the control flow tree node number is prefixed by the letter P, this indicates that the statement is a procedure call and the node specified is the first executed. TRIANGLE_COMPLETE has a considerably higher control flow complexity than any of the previous procedures tested. Further, the RIGHT_ANGLE_CHECK procedure, which has the hardest predicates to satisfy, is the last for control to encounter. Consequently, for a point to reach this procedure it must satisfy 9 predicates, or in other words, not deviate from 9 branches.

```
Control flow tree node number
     Line number
          Ada statements

       1  procedure TRIANGLE_COMPLETE ( A,B,C in INTEGER; TRI_KIND out TRI_TYPE ) is
       2   P : INTEGER;
       3   procedure TRIANGLE_2 ( A,B,C in INTEGER; TRI_KIND out TRI_TYPE ) is
       4    procedure RIGHT_ANGLE_CHECK ( A,B,C in INTEGER; TRI_KIND out TRI_TYPE ) is
       5    begin
 20 [  6     if ( ( ( A * A ) + ( B * B ) ) = ( C * C ) ) then
 23 [  7      TRI_KIND := RIGHT_ANGLED_SCALENE;
       8     else
 22 [  9      if ( ( ( B * B ) + ( C * C ) ) = ( A * A ) ) then
 25 [ 10       TRI_KIND := RIGHT_ANGLED_SCALENE;
      11      else
 24 [ 12       if ( ( ( A * A ) + ( C * C ) = ( B * B ) ) then
 27 [ 13        TRI_KIND := RIGHT_ANGLED_SCALENE;
      14       else
 26 [ 15        TRI_KIND := NON_RIGHT_ANGLED_SCALENE;
      16       end if;
      17      end if;
      18     end if;
      19    end RIGHT_ANGLE_CHECK;
      20   begin
 13 [ 21    if ( A = B ) then
 15 [ 22     If ( B = C ) then
 19 [ 23      TRI_KIND := EQUILATERAL;
      24     else
 18 [ 25      TRI_KIND := ISOSCELES;
      26     end if;
      27    else
 14 [ 28     if ( A = C ) then
 17 [ 29      TRI_KIND := ISOSCELES;
      30     else
 16 [ 31      if ( B = C ) then
 21 [ 32       TRI_KIND := ISOSCELES;
      33      else
P20 [ 34       RIGHT_ANGLE_CHECK ( A, B, C, TRI_KIND );
      35      end if;
      36     end if;
      37    end If;
      38   end TRIANGLE_2;
      39  begin
  1 [ 40   if ( A > 0 ) then
  3 [ 41    if ( B > 0 ) then
  5 [ 42     if ( C > 0 ) then
  7 [ 43      P := ( A + B + C );
    [ 44      if ( ( 2 * A ) < P ) then
  9 [ 45       if ( ( 2 * B ) < P ) then
 11 [ 46        if ( ( 2 * C ) < P ) then
P13 [ 47         TRIANGLE_2 ( A, B, C, TRI_KIND );
      48        else
 12 [ 49         TRI_KIND := NOT_A_TRIANGLE;
      50        end if;
 10 - 51       end if;
  8 - 52      end if;
      53     else
  6 [ 54      TRI_KIND := NOT_A_TRIANGLE;
      55     end if;
      56    else
  4 [ 57     TRI_KIND := NOT_A_TRIANGLE;
      58    end If;
      59   else
  2 [ 60    TRI_KIND := NOT_A_TRIANGLE;
      61   end if;
      62  end TRIANGLE_COMPLETE;
```

Figure 6.12 - The TRIANGLE_COMPLETE procedure

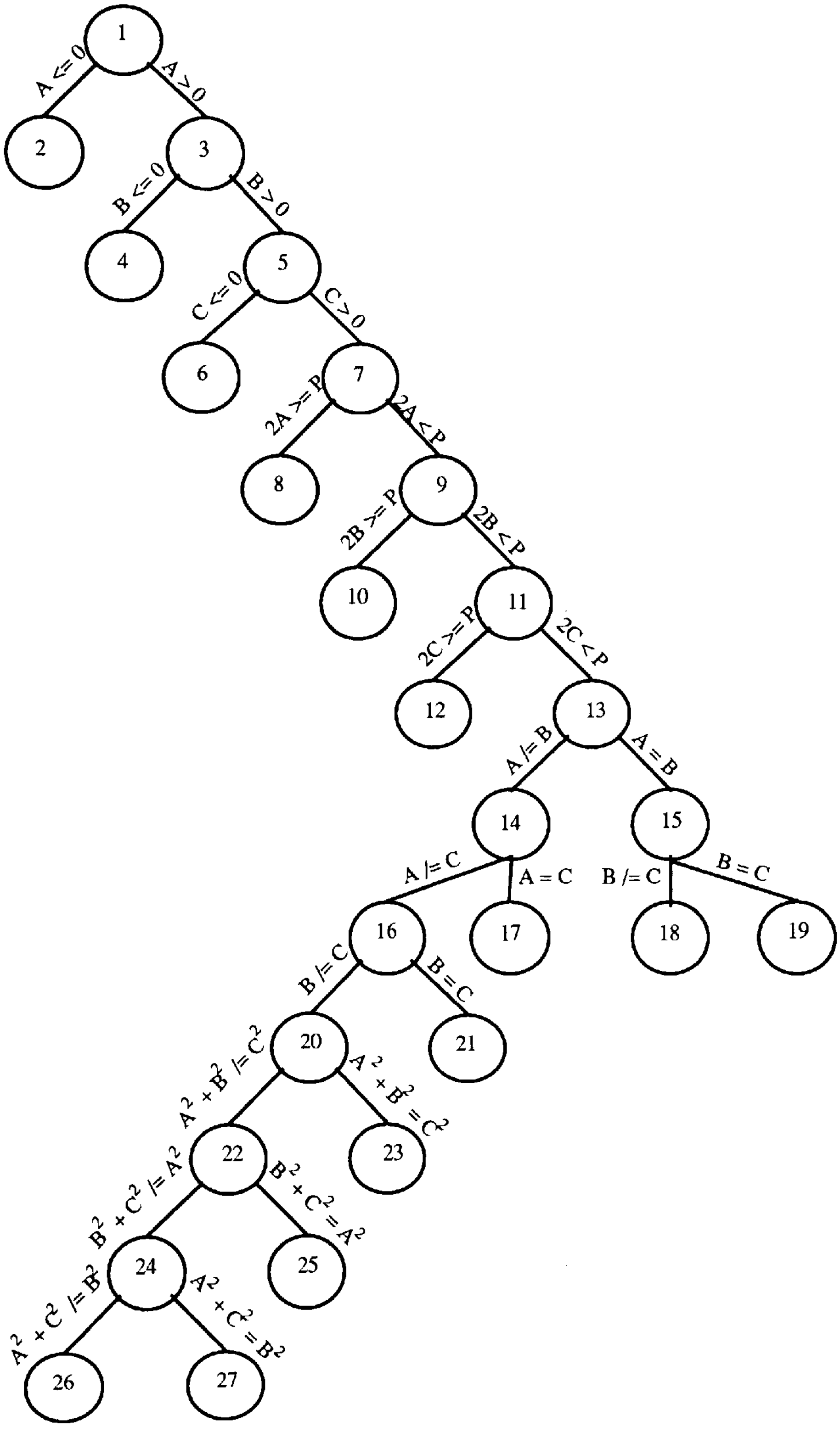


Figure 6.13 - Control flow tree of the TRIANGLE_COMPLETE procedure

Table 6.31 contains the initial points for the three HATS harness runs. The initial points for runs T10 and T11 were specifically selected to traverse the significant partial path to RIGHT_ANGLE_CHECK so that the heuristics would consider other nodes in RIGHT_ANGLE_CHECK without the possibility of the HATS harness deeming this part of the control flow tree infeasible.

HATS	Input variable		
run	A	B	C
T10	3	5	4
T11	3	4	5
T12	56	23	19

Table 6.31 -TRIANGLE_COMPLETE procedure's initial points

6.6.1 Run T10

Table 6.32 contains the traversal results for this run.

Initial point path	1,3,5,7,9,11,13,14,16,20,22,24,27		
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	3	10
4	DA	3	12
6	DA	3	
8	LP	3	17
15	LP	3	26,18
19	LP	3	
21	LP	3	
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
23	LP	4	
"	BF	10	
Nodes unconsidered			
Considered node	Nodes unconsidered		
23	25		

Table 6.32 - Run T10 traversal results

With nodes 8, 15, 19 and 21 the LP chose a start point that is very close to the solution point located. On node 23, the LP terminated since upper-deviations were produced after modifying each input variable. BF then took over and failed after only 10 iterations since it could not allocate the Follow role to any of the input variables. This was considered unusual since the BF is the most suitable heuristic, and deserves further investigation.

	9	*	*	*	*	*	*	*	*	101
	8	*	*	*	*	*	*	*	73	84
	7	*	*	*	*	*	*	*	58	69
	6	*	*	*	*	*	29	*	45	*
B	5	*	*	*	*	13	18	*	*	45
	4	*	*	*	*	*	*	*	*	*
	3	*	*	*	*	-3	*	*	18	29
	2	*	*	*	*	*	-3	*	13	*
	1	*	*	*	*	*	*	*	*	*
		-2	-1	0	1	2	3	4	5	6
C = 4						A				

Figure 6.14 - Run T10 (A, B) partial input plane : BF considering node 23 in the Determine-initial-follow-and-cross-details phase

The BF commences the Determine-initial-follow-and-cross-details phase with a central point of (2, 5, 4). Variable A is allocated the Cross role since its modification produced a positive-sibling-traversal (3, 5, 4) and an upper-deviation (1, 5, 4) (figure 6.14).

However, BF could not allocate the Follow role since modifying the remaining two input variables B ((2, 6, 4) and (2, 4, 4)) and C (2, 5, 5) and (2, 5, 3)) produced only upper-deviations (figure 6.15). BF requires that at least one of the modifications to an input variable, produces a sibling-traversal for the Follow role to be allocated to the variable.

	8	*	*	*	*	*	*	-11	*	21
	7	*	*	*	*	*	-9	*	19	*
	6	*	*	*	*	-7	*	17	*	*
	5	*	*	*	-5	*	15	*	*	*
C	4	*	*	-3	*	13	*	*	*	*
	3	*	*	*	11	*	*	*	*	*
	2	*	*	*	*	*	*	*	*	*
	1	*	*	*	*	*	*	*	*	*
	0	*	*	*	*	*	*	*	*	*
		1	2	3	4	5	6	7	8	9
A = 2						B				

Figure 6.15 - Run T10 (B, C) partial input plane : BF considering node 23 in the Determine-initial-follow-and-cross-details phase

Figures 6.14 and 6.15 show a much more partitioned sibling-traversal domain to that seen previously. This is due to the branches above node 23.

Let us consider the three 2 dimensional planes around the central point (2, 5, 4), these being (A, B) (figure 6.14), (A, C) and (B, C) (figure 6.15). When the BF allocates the Follow and Cross roles to two variables, it is committed to follow a boundary in the corresponding plane. Both the (A, B) (figure 6.14) and (A, C) planes have isolated sibling-traversal domains around the central point with no solutions present. Had the BF followed a boundary in one of these planes then it may end up circling the same sibling-traversal domain. In the (B, C) plane (figure 6.15) there are two sibling-traversal domains, the left-most starting at point (2, 3, 4) and the other at point (2, 4, 3). An exhaustive search further along these sibling-traversal domains, revealed that no solutions were present, hence following the boundary surrounding them would be pointless. There is a possibility that this could be inferred since the predicate values shown in figure 6.15 progress away from 0.

To establish if a solution point does exist and where, a three dimensional exhaustive search was conducted around the central point. This spanned the range (-2, 1, 0) to (6, 9, 8). Two solutions were found at points (3, 4, 5) and (4, 3, 5), which are very close to the central point. In its present form, the BF could not locate either of these points since it can only move in the same two dimensions. Further, the central point chosen determines which boundaries the BF may follow. Hence, this also influences the success of the BF.

6.6.2 Run T11

Table 6.33 contains the traversal results for this run.

Initial point path		1,3,5,7,9,11,13,14,16,20,23	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	3	12
4	DA	3	
6	DA	3	10
8	LP	3	15,18
17	LP	6	21,22,24,26
"	BF	5	19
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
25	LP	4	
"	BF	496	
Nodes unconsidered			
Considered node		Nodes unconsidered	
25		27	

Table 6.33 - Run T11 traversal results

The LP failed on linear node 17 when there are solutions surrounding the start point of (3, 4, 5). Both figures 6.16 and 6.17 show that the start point is partially surrounded by upper-deviations, which cause the LP to terminate. The BF succeeds during its Obtain-a-close-point phase, since it used a different but closer start point to the expected boundary, which was generated during the LP's attempt.

	7	*	*	-2	-1	0	1	*
	6	*	-3	-2	-1	0	*	2
	5	-4	-3	-2	-1	*	1	2
B	4	*	-3	-2	*	0	1	2
	3	*	*	*	-1	0	1	2
	2	*	*	*	-1	0	1	*
	1	*	*	*	*	0	*	*
		1	2	3	4	5	6	7
C = 5					A			

Figure 6.16 - Run T11 (A, B) partial input plane : LP considering node 17

	7	*	*	*	*	-4	-4	-4
	6	*	*	*	-3	-3	-3	-3
	5	*	*	*	-2	-2	-2	-2
C	4	*	-1	*	-1	-1	-1	*
	3	0	0	*	0	0	*	*
	2	*	1	*	1	*	*	*
	1	*	*	*	*	*	*	*
		1	2	3	4	5	6	7
A = 3					B			

Figure 6.17 - Run T11 (B, C) partial input plane : LP considering node 17

On node 25, the BF selected the central point (3, 5, 4) and allocated the Follow role to variable B and Cross to A. After following the boundary for a little way the Reorient-boundary-follower phase was invoked and the initial allocations swapped. The BF continued till the node iteration threshold was reached and terminated. No solution was found and deserves further investigation.

The area around the boundary followed (in the range 0 to 500 for A and B, with C constant at 4) was exhaustively searched. Only one solution was found at point (5, 3, 4), which is very close to the central point. Had the boundary been followed in the opposite direction then this solution point would have been found.

A larger, exhaustive, three dimensional search in the range 0 to 500 for each input variable (total of 501^3 points) was conducted and 772 solutions to node 25 were found. Yet there is only one solution when C is held constant at 4. Clearly, restricting the BF to operate in only two dimensions can severely limit the number of potential solutions. However, increasing the dimensionality of the search also increases the number of non-solution points.

6.6.3 Run T12

Table 6.34 contains the traversal results for this run.

Initial point path			1,3,5,7,8
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
2	DA	3	9,10
4	DA	3	
6	DA	3	
11	LP	9	13,14,16.21
12	LP	4	
15	LP	6	19
17	LP	5	20,22,24,26
Nodes considered and untraversed			
Node	Heuristic	Iters	Coincidental nodes
23	LP	4	18
"	BF	496	
Nodes unconsidered			
Considered node		Nodes unconsidered	
23		25,27	

Table 6.34 - Run T12 traversal results

On node 23 the BF chose a start point of (18, 19, 20) which has a predicate value of 285. Using the central point (18, 19, 26), variable C was allocated the Follow role and variable B, Cross. Table 6.35 shows the operations from the commencement of the Follow-boundary phase. Notice that the boundary located was followed by mainly increasing variable C. Figure 6.18 shows the domain operated in by the BF in table 6.35. Notice that the predicate values decrease and continue to decrease as variable C increases.

Iterations 23, 27, 28, 31 and 32 show the predicate value decreasing as the boundary located is followed. This continues right up to the last point causing sibling-traversal (iteration 492 of 496) before the node iteration threshold is reached.

Iter	Var	Phase	Generate point			Execute point			Trav	Pred	Next
			A	B	C	A	B	C	effect	value	action
23	C	FB F	18	19	26	18	19	27	-ST	-44	C 1
24	B	FB C	18	19	27	18	20	27	-ST	-5	C 1
25	B	FB C	18	19	27	18	18	27	UD		F
26	C	FB F	18	18	27	18	18	28	UD		C 1
27	B	FB C	18	18	28	18	19	28	-ST	-99	F
28	C	FB F	18	19	28	18	19	29	-ST	-156	C 1
29	B	FB C	18	19	29	18	18	29	UD		F
30	C	FB F	18	18	29	18	18	30	UD		C 1
31	B	FB C	18	18	30	18	19	30	-ST	-215	F
32	C	FB F	18	19	30	18	19	31	-ST	-276	C 1

Table 6.35 - Run T12 excerpt : BF considering node 23 in the first Follow-boundary phase

	35	*	*	*	-540	-501
	34	*	-543	*	-471	-432
	33	-509	-476	*	-404	-365
	32	-444	-411	*	-339	-300
	31	-381	-348	*	-276	-237
	30	-320	-287	*	-215	-176
	29	-261	-228	*	-156	-117
	28	-204	-171	*	-99	-60
C	27	-149	-116	*	-44	-5
	26	-96	-63	*	9	48
	25	-45	-12	*	60	99
	24	4	37	*	109	148
	23	51	84	*	156	195
	22	96	129	*	201	240
	21	139	172	*	244	283
	20	180	213	*	285	*
	19	219	252	*	*	363
		16	17	18	19	20
A=18				B		

Figure 6.18 - Run T12 (B, C) partial input plane : BF considering node 23; from the heuristic's start point

The BF has followed a boundary but it does not take into consideration that the predicate value is ever decreasing. A predicate value moving in mainly one direction away from 0, indicates that there is a progressively reducing chance of finding a solution. Ideally, as the boundary is followed there would be a healthy alternation between positive and negative predicate values or upper-deviations and small predicate values. However, solutions can still be found in less than ideal situations. Table 6.35 and figure 6.18 illustrate that the BF has followed the boundary defined by upper-deviations and negative-sibling-traversals from the central point upwards in figure 6.18. An exhaustive search of the input space, in the range 0 to 500 for variables B and C with A constant at 18, revealed two solution points (18, 24, 30) and (18, 80, 82). The first solution is close to the central point. However, the BF did not locate it as it was following a different boundary. Figure 6.19 shows the domain surrounding the central point and the first solution point. There are two boundaries in this partial input plane, the first is the vertical line of upper-deviation points. The second is the notional boundary which exists between the positive and negative predicate values and surrounds the solution point. Notice that the notional boundary emanates from the real boundary.

	31	-348	*	-276	-237	-196	-153	-108	-61	-12
	30	-287	*	-215	-176	-135	-92	-47	0	49
C	29	-228	*	-156	-117	-76	-33	12	59	108
	28	-171	*	-99	-60	-19	24	69	116	165
	27	-116	*	-44	-5	36	79	124	171	220
	26	-63	*	9	48	89	132	177	224	273
		17	18	19	20	21	22	23	24	25
A=18						B				

Figure 6.19 - Run T12 (B, C) partial input plane : BF considering node 23; showing part of the boundary followed and a solution point

The central point lies close to two boundaries. The BF is unable to determine when this is the case and so make a decision over which boundary to follow.

6.6.4 Comparison of HATS with Random Testing on the TRIANGLE_COMPLETE Procedure

The random testing range for each input variable is ± 100, which is the same as HATS's initial point selection range. Each random testing run continues till all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs T10 to T12.

The results for nodes 2 to 13 (table 6.36) have little difference to the isolated TRIANGLE procedure's (section 6.3.4). On nodes 14 to 21, HATS performance is considerably better than random and has little difference to that on the isolated TRIANGLE_2 procedure (section 6.4.4). However, random takes between two and five times more iterations to those in section 6.4.4. This is due to the TRIANGLE procedure reducing the solution domain size and increasing the potential for control deviation. The difference between HATS and random on node 19 is particularly notable; 3 : 83699.1.

Nodes 22, 24 and 26 are easy nodes, which HATS covers in less iterations than random. On nodes 23, 25 and 27, HATS is unable to locate a solution. However, complete coverage is normally achieved when the RIGHT_ANGLE_CHECK is considered in isolation. This illustrates the impact of increased control and partial path function complexity to the considered node. Random testing effort is considerably increased from section 6.5.4.

Node	HATS iterations	Random iterations
1	IPT	1
2	3	1.98
3	IPT/CT	2.034
4	3	3.852
5	IPT/CT	4.42
6	3	8.458
7	IPT/CT	8.62
8	3	47.768
9	IPT/CT	10.318
10	IPT/CT	50.964
11	9	12.376
12	4	47.176
13	IPT/CT	16.278
14	IPT/CT	16.446
15	3-6	1072.03
16	IPT/CT	16.554
17	5-11	1119.62
18	IPT/CT	1080.77
19	3	83699.1
20	IPT/CT	16.752
21	3	1037.8
22	IPT/CT	16.752
23	NNT	84072.5
24	IPT/CT	16.752
25	NNT	80568.2
26	IPT/CT	16.752
27	NNT	79940.3

Key :

IPT	Initial point traversed
NNT	Node not traversed
CT	Coincidentally traversed

Table 6.36 - The number of iterations taken by HATS and random testing for each of TRIANGLE_COMPLETE's nodes

6.7 Overall Discussion

Table 6.37 shows the overall branch coverage for each procedure and all procedures.

Procedure	Branches in procedure	Branches covered out of total for all runs	Branch coverage %
TRIANGLE	12	36/36	100
TRIANGLE_2	8	24/24	100
RIGHT_ANGLE_CHECK	6	16/18	89
TRIANGLE_COMPLETE	26	71/78	91
All procedures	52	147/156	94

Table 6.37 - HATS branch coverage on the triangle related procedures

With RIGHT_ANGLE_CHECK 100% coverage was not achieved as the BF pursued a boundary not knowing that no solutions exist upon it. With TRIANGLE_COMPLETE, full coverage was not achieved as the BF was adversely affected by upper-deviations and the BF pursued boundaries not knowing that no solutions exist upon it or in the direction being followed. Clearly, increasing the number of branches and non-linear predicates, especially involving equality, increases the number of iterations required. HATS finds it particularly difficult when non-linear equality predicates are in the lower region of a procedure's significant control flow tree. The performance of each heuristic is now discussed.

6.7.1 Direct Assignment Heuristic

The DA worked well without any problems and generated some useful points which were used by the LP and BF.

6.7.2 Linear Predictor Heuristic

Generally the LP worked well. However when upper-deviations were produced, this caused some difficulty, resulting in additional iterations being used and may even cause the LP to terminate. More careful management of the closest point to the expected boundary is required. The LP's termination criteria worked well, operating at the earliest time and not while there was potential for a solution to be found. The LP has made efficient use of iterations due to its direct nature of solution location and effective termination criteria. However, better handling of upper-deviations is required.

6.7.3 Boundary Follower Heuristic

When possible the BF has accurately followed a boundary. However, the RIGHT_ANGLE_CHECK and TRIANGLE_COMPLETE procedures have raised a number of concerns. The BF can follow a boundary in the opposite direction to a solution or where no solution exists. This indicates two areas for improvement. First, determining a direction to follow a boundary. Second, introducing termination criteria to ensure that boundary following continues only when there is a good chance of a solution existing. Related to this second point is the ability to prevent such a boundary from being followed. It should not be the responsibility of the node iteration threshold to play a dual role and operate as a secondary heuristic termination criterion as well as an upper iteration limit.

7 The Remainder and Linear Search Problems

7.1 Introduction

This chapter introduces two new testing problems to HATS; loops and the composite data type, arrays. Two procedures are used as vehicles. The first, a remainder procedure, has a number of loops and the second, a linear search procedure, combines both a loop and an integer array. These new problems and the enhancements to HATS in order to test them are described. The DA, LP and BF heuristics are applied to both the procedures. Results and discussion are presented for branch and mutation testing of the remainder procedure and branch testing of the linear search procedure.

7.2 The REMAINDER Procedure

The REMAINDER procedure (figure 7.1) calculates the remainder after an integer division. REMAINDER has two integer input variables; A, the dividend, and B, the divisor. The single output variable, REM, is integer and is the remainder after A has been divided by B. All conditions within the procedure have a linear relationship with the input variables. There are 18 branches in the procedure and the control flow tree (figure 7.2) has 19 nodes, with the longest path consisting of 6 nodes. The changes to the control flow tree necessary to represent loops are described in the next section.

Table 7.1 shows the six cases handled in the procedure. Cases 3 to 6 are handled by the four loops in the procedure, where the remainder is computed. Case 2, division by zero, has been simplified to ease the management of exceptions.

Case	Value of A	Value of B	Valid	Value of REM	Handled by node(s)
1	0	any value	yes	0	3
2	/= 0	0	no	0	5
3	-ve	-ve	yes	remainder	12,13
4	-ve	+ve	yes	remainder	14,15
5	+ve	-ve	yes	remainder	16,17
6	+ve	+ve	yes	remainder	18,19

Table 7.1 - The 6 cases handled in the REMAINDER procedure

```
┌ Control flow tree node number
│  ┌ Line number
│  │     ┌ Ada statements
↓  ↓     ↓
     1  procedure REMAINDER ( A, B in INTEGER; REM out INTEGER ) is
     2    N,R : INTEGER;
     3  begin
 1┌  4    R := 0;
  │  5    N := 0;
  └  6    if ( A = 0 ) then
 3[  7      REM := 0;
     8    else
 2[  9      if ( B = 0 ) then
 5[ 10        REM := 0;
    11      else
 4[ 12        if ( A > 0 ) then
 7[ 13          if ( B > 0 ) then
11┌ 14            R := A;
  └ 15            while ( ( A - N ) >= B ) loop
18┌ 16              N := N + B;
  └ 17              R := A - N;
    18            end loop;
    19          else
10┌ 20            R := A;
  └ 21            while ( ( A + N ) >= abs ( B ) ) loop
16┌ 22              N := N + B;
  └ 23              R := A + N;
    24            end loop;
    25          end if;
    26        else
 6[ 27          if ( B > 0 ) then
 9┌ 28            R := A;
  └ 29            while ( abs ( A + N ) >= B ) loop
14┌ 30              N := N + B;
  └ 31              R := A + N;
    32            end loop;
    33          else
 8┌ 34            R := A;
  └ 35            while ( ( A - N ) <= B ) loop
12┌ 36              N := N + B;
  └ 37              R := A - N;
    38            end loop;
    39          end if;
    40        end if;
    41      end if;
    42    end if;
    43    REM := R;
    44  end REMAINDER;
```

Figure 7.1 - The REMAINDER procedure

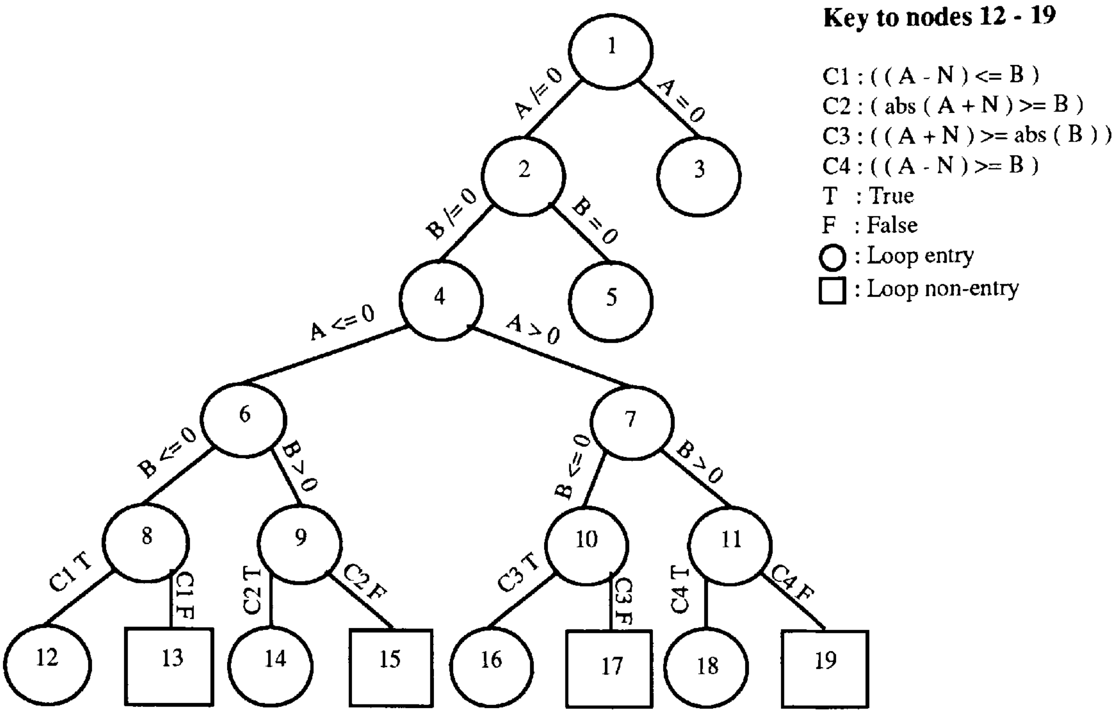


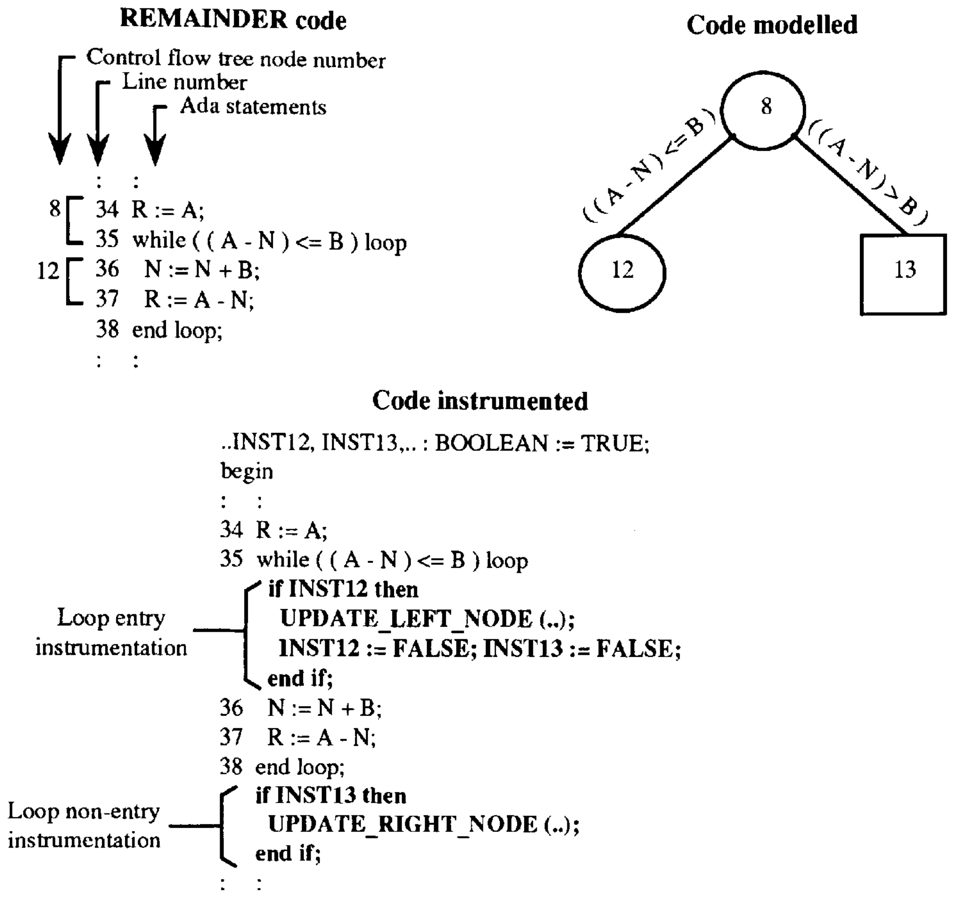
Figure 7.2 - Control flow tree of the REMAINDER procedure

7.3 Techniques for Testing Loops

Several issues must be considered before loops can be tested in HATS. Exactly what constitutes branch testing of loops must be established. A loop has two branches; one where control enters the loop and iterates the loop one or more times, and the other where control does not enter the loop, hence the loop iterates zero times. Therefore, to satisfy branch testing, a point should cause the loop's condition to be true upon its first evaluation, and a further point should cause the condition to be false upon its first evaluation. The order of entry and non-entry to a loop does not matter.

Although the present program model, a control flow (binary) tree, does not allow any cycles, it is possible to represent a loop's branches as nodes (figure 7.2). A circular node represents entry to the loop and the sequence of non-control flow affecting statements the loop contains. Statements within the loop are shown as nodes emanating from the loop's circular node. A square node represents non-entry to the loop. This node's number is not shown in the procedure text (figure 7.1) since the jump it corresponds to is not explicitly shown in the procedure text.

Instrumenting loops presents two distinct concerns. Firstly, if data were recorded on every iteration of a loop then this would result in significantly more data being stored in the program model. However, this is unnecessary as the heuristics only require data to be recorded when control first encounters the loop. The second concern is recording non-entry to the loop. Unlike an "if" statement without an "else" where the "else" can be inserted for instrumentation purposes to record the false branch, this is not possible with a loop. The solution to these two concerns is to control when the instrumentation works. The instrumentation will only "fire" if a Boolean variable, or instrumentation switch, is true. Figure 7.3 shows how one of the while loops in the REMAINDER procedure is instrumented.



```
    :  :
8   34 R := A;
    35 while ( ( A - N ) <= B ) loop
12  36   N := N + B;
    37   R := A - N;
    38 end loop;
    :  :
```

```
..INST12, INST13,.. : BOOLEAN := TRUE;
begin
:  :
34 R := A;
35 while ( ( A - N ) <= B ) loop
     if INST12 then
       UPDATE_LEFT_NODE (..);
       INST12 := FALSE; INST13 := FALSE;
     end if;
36   N := N + B;
37   R := A - N;
38 end loop;
   if INST13 then
     UPDATE_RIGHT_NODE (..);
   end if;
:  :
```

Figure 7.3 - The modelling and instrumentation of a loop

All instrumentation switches are initialised to true. The instrumentation to record entry to the loop is placed as the first statements inside the loop. If control enters the loop, the instrumentation (INST12) will fire which updates the control flow tree and sets the switches for this instrumentation (INST12) and the instrumentation for non-entry to the loop (INST13), to false. This ensures that the control flow tree is updated once when the loop is entered and is not updated by the loop non-entry instrumentation when the loop stops iterating. The instrumentation for non-entry to the loop does not affect any instrumentation switches.

The program model proposed above could be considered as a simplification of the exemplar-path tree model for structural testing (Cimittle and Carlini, 1991).

7.4 REMAINDER Branch Testing Experiments

7.4.1 HATS Experimental Set Up

Five initial points have been hand selected in the range ± 20 for both input variables (table 7.2).

HATS	Input variable	
run	A	B
R1	-6	-3
R2	19	-4
R3	-18	5
R4	10	3
R5	3	10

Table 7.2 - REMAINDER procedure's initial points

The points were selected so that each loop is covered by at least one point. This enables the heuristics to consider all the loops over the five runs. The node iteration threshold is 500 and the LP and BF consider input variables in the order A first then B, second.

7.4.2 Run R1

Table 7.3 contains the traversal results for this run.

Initial point path		1,2,4,6,8,12	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	13,7,10,17
5	DA	3	9,14
11	DA	3	16,18
15	LP	8	
19	LP	3	

Table 7.3 - Run R1 traversal results

	3	5	4	3	2	1	0	-1	-2	*	*
	2	6	5	4	3	2	1	0	-1	*	*
B	1	7	6	5	4	3	2	1	0	*	*
	0	*	*	*	*	*	*	*	*	*	*
	-1	*	*	*	*	*	*	*	*	*	*
		-8	-7	-6	-5	-4	-3	-2	-1	0	1
						A					

Figure 7.4 - Run R1 partial input space : LP considering node 15

On node 15, the LP starts from point (-6, 1), generated by the DA on node 5, and was unable to locate a solution increasing variable A (figure 7.4). The closest point the LP located (-1, 1), is the corner point of the domain satisfying node 9 (node 15's parent). From this point, B is increased to locate the solution point (-1, 2).

On node 19, the LP takes three iterations over finding a solution point in the Determine-linearity phase.

7.4.3 Run R2

Table 7.4 contains the traversal results for this run.

Initial point path			1,2,4,7,10,16
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	6,8,13,17
5	DA	3	11,18
9	DA	3	12,9,14
15	LP	2	
19	LP	8	

Table 7.4 - Run R2 traversal results

Node 15's solution point was found during the LP's Determine-linearity phase. On node 19, the LP starts from point (19, 1), generated by the DA on node 5, and was unable to locate a solution decreasing variable A (figure 7.5). The closest point the LP located (1, 1) is the corner point of the domain satisfying node 11 (node 19's parent). From this point, B is increased to locate the solution point (1, 2).

	3	*	*	-2	-1	0
	2	*	*	-1	0	1
B	1	*	*	0	1	2
	0	*	*	*	*	*
	-1	*	*	*	*	*
		-1	0	1	2	3
				A		

Figure 7.5 - Run R2 partial input space : LP considering node 19

7.4.4 Run R3

Table 7.5 contains the traversal results for this run.

Initial point path			1,2,4,6,9,14
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	15,7,11,19
5	DA	3	12,8
10	DA	3	16,18
13	LP	8	
"	BF	6	
17	LP	3	
"	BF	6	

Table 7.5 - Run R3 traversal results

On two nodes the LP was unsuccessful, resulting in the application of the BF. This was unexpected as the REMAINDER has only linear nodes. On node 13, the LP starts from point (-18, -1), generated by the DA on node 5, and was unable to locate a solution increasing variable A (figure 7.6) since an upper-deviation is produced. The closest point the LP located (-1, -1) is the corner point of the domain satisfying node 8 (node 13's parent). From this point, variable B is increased producing an upper-deviation which causes the LP to terminate.

BF starts from the point (-1, -1) and is unable to locate a closer point in the Obtain-a-close-point phase since two upper-deviations were produced. During the Determine-initial-follow-and-cross-details phase a solution is located at (-1, -2).

	1	*	*	*	*	*
B	0	*	*	*	*	*
	-1	-2	-1	0	*	*
	-2	-1	0	1	*	*
		-3	-2	-1	0	1
				A		

Figure 7.6 - Run R3 partial input space : LP and BF considering node 13

On node 17, the LP starts from point (1, -1), generated by the DA on node 10, and terminates after producing upper-deviations modifying both input variables. The closest point located by the LP is (1, -1) (figure 7.7). The BF is unable to locate a closer point and locates a solution point (1, -2) during the Determine-initial-follow-and-cross-details phase.

	1	*	*	*	*	*
	0	*	*	*	*	*
B	-1	*	*	0	1	2
	-2	*	*	-1	0	1
	-3	*	*	-2	-1	0
		-1	0	1	2	3
				A		

Figure 7.7 - Run R3 partial input space : LP and BF considering node 17

7.4.5 Run R4

Table 7.6 contains the traversal results for this run.

Initial point path		1,2,4,7,11,18	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	6,9,15,19
5	DA	3	10,16
8	DA	3	12,14
13	LP	2	
"	BF	6	
17	LP	8	
"	BF	6	

Table 7.6 - Run R4 traversal results

On node 13, the LP starts from point (-1, -1), and produces upper-deviations modifying both input variables, then terminates. The BF takes over, starting from point (-1, -1) and finds a solution point at (-1, -2).

On node 17, the LP starts from point (10, -1), generated by the DA on node 5, and terminates after producing upper-deviations modifying both input variables. The closest point the LP could locate is (1, -1). The BF does not locate a closer point to start from and goes on to locate a solution point at (1, 2).

7.4.6 Run R5

Table 7.7 contains the traversal results for this run.

Initial point path		1,2,4,7,11,19	
Nodes considered and traversed			
Node	Heuristic	Iters	Coincidental nodes
3	DA	3	6,9,15
5	DA	3	10,16,18
8	DA	3	12,14
13	LP	2	
"	BF	6	
17	LP	8	
"	BF	6	

Table 7.7 - Run R5 traversal results

The consideration of node 13 follows exactly the same operational pattern as the same node in the previous run (R4). The LP starts from point (-1, -1) and the BF locates a solution point at (1, 2).

On node 17, the LP starts from point (3, 1) and terminates after producing upper-deviations on both input variables. The closest point the LP could locate is (1, -1). The BF is unable to locate a closer point to start from and locates a solution point (1, -2) during the Determine-initial-follow-and-cross-details phase. The BF's operational pattern is exactly the same as in runs R3 and R4.

7.4.7 Overall Discussion for the HATS Branch Testing Runs

Two phenomena have been observed in the HATS branch testing runs. Firstly, repeated heuristic operation patterns and secondly, the LP's inability to locate a solution, despite being close to one.

The REMAINDER's input space is divided into 10 partitions (figure 7.8). The first two partitions correspond to the first two cases outlined in table 7.1. Cases 3 to 6 are defined by the REMAINDER's four while loops and are shown in figure 7.8 as quadrants. Each while loop's condition divides each quadrant into two. One half for entry to the corresponding while loop and one for non-entry, making the remaining 8 input space partitions. In other words, each partition corresponds to the path to each leaf node in the control flow tree (figure 7.2).

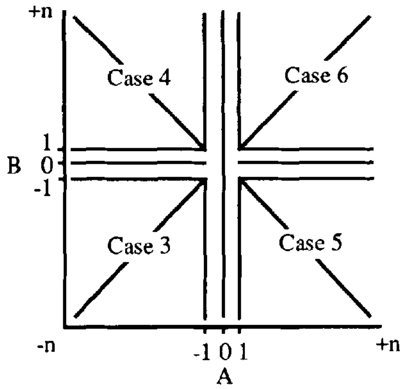


Figure 7.8 - REMAINDER procedure's input space

When the DA is applied to nodes 2, 3, 4 or 5, values of -1, 0, and 1 are produced for variable A or B. The execute point traverses the considered node and is just inside one of the quadrants where control arrives at a while loop. If one of the loop's nodes is untraversed then this node is considered by the LP. The LP converges on the corner point since it modifies one input variable at a time. Figures 7.4 to 7.7 show that the corner point has a predicate value of 0. When the LP has located a quadrant's corner point and the considered node remains untraversed, Linearity-determination may either locate a solution point or cause an upper-deviation. The outcome is dependent upon which node is considered (the quadrant currently in) and which input variable is about to be increased by 1. A solution point was located in run R1, node 15 and run R2, node 19. However, in other runs, i.e. run R3, nodes 13 and 17, an upper-deviation was produced resulting in the LP's termination. The BF takes over and does not find solution points in the Obtain-a-close-point phase, but does in the Determine-initial-follow-and-cross-details phase. This is because the corner point was used as the central point and the phase increases and decreases each input variable.

Repeated heuristic operation patterns have been observed due to the heuristics using the same points in different runs. Because the heuristics are deterministic the same actions will be taken and points generated.

The two phenomena identified are due to the REMAINDER's input space and the points produced by the DA.

7.4.8 Comparison of HATS with Random Testing on the REMAINDER Procedure

The random testing range for each input variable is ± 20, which is the same as HATS's initial point selection range. Each random testing run continues until all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs R1 to R5.

Node	HATS iterations	Random iterations
1	IPT	1
2	IPT/CT	1.03
3	3	43.666
4	IPT/CT	1.058
5	3	42.20
6	IPT/CT	2.096
7	IPT/CT	2.272
8	3	4.316
9	3	4.308
10	3	4.404
11	3	4.21
12	IPT/CT	7.88
13	8-14	8.842
14	IPT/CT	7.762
15	2-8	9.314
16	IPT/CT	8.18
17	9-14	9.402
18	IPT/CT	7.906
19	3-8	8.652

Key :

IPT	Initial point traversed
NNT	Node not traversed
CT	Coincidentally traversed

Table 7.8 - The number of iterations taken by HATS and random testing for each of REMAINDER's nodes

Table 7.8 shows that nodes 2, 4, 6 and 7 have the largest solution domain and are consequently the easiest to traverse. Nodes 3 and 5 have the smallest solution domains in the REMAINDER procedure. On these nodes, HATS takes 3 iterations to random's 43 iterations (approximately). On nodes 8 to 11 the iterations taken by both HATS and random are similar. Nodes 12 to 19 have similar domain sizes. With HATS the nodes with the slightly larger solution domains (entry to loop) are covered by the initial point or by coincidental traversal. Again, the iterations taken by both is similar.

7.5 Mutation Analysis of the REMAINDER Procedure

Budd (1981) outlines four levels of mutation analysis; statement, predicate, domain and coincidental correctness; levels 1 to 4 respectively. Levels 1 and 2 were applied to the QUADRATIC procedure (chapter 4). However, it is unnecessary to produce level 1 (statement analysis) mutants as this is achieved by HATS's branch testing criterion.

Predicate analysis consists of three mutation operators; absolute operator insertion, relational operator alteration and predicate alteration by a small amount. To further reduce the number of mutants, only predicate alteration by a small amount will be used. This mutation operator produces mutants that are the hardest of the three to reveal since it causes only a small change in the location of input space boundaries. The other two mutation operators generally have a more substantial effect on the input space, which is easier to detect.

7.5.1 Mutation Experiments

Applying the mutation operator, predicate alteration by a small amount, to the REMAINDER procedure produces 38 mutants (table 7.9).

Mutant	Line no	Original condition	Mutated condition	Simplified mutation	Identical to	Used
RM1	6	(A=0)	(A-1)=0	(A=1)		Y
RM2	6	"	(A+1)=0	(A=-1)		Y
RM3	9	(B=0)	(B-1)=0	(B=1)		Y
RM4	9	"	(B+1)=0	(B=-1)		Y
RM5	12	(A>0)	(A-1)>0	(A>1)		Y
RM6	12	"	(A+1)>0	(A>-1)	Original	N
RM7	13	(B>0)	(B-1)>0	(B>1)		Y
RM8	13	"	(B+1)>0	(B>-1)	Original	N
RM9	15	((A-N)>=B)	((A-1)-N)>=B		RM12, RM14	Y
RM10	15	"	((A+1)-N)>=B		RM11, RM13	Y
RM11	15	"	(A-(N-1))>=B		RM10, RM13	N
RM12	15	"	(A-(N+1))>=B		RM9, RM14	N
RM13	15	"	(A-N)>=(B-1)		RM10, RM11	N
RM14	15	"	(A-N)>=(B+1)		RM9, RM12	N
RM15	21	((A+N)>=abs(B)	((A-1)+N)>=abs(B)		RM17, RM22	Y
RM16	21	"	((A+1)+N)>=abs(B)		RM18, RM21	Y
RM17	21	"	(A+(N-1))>=abs(B)		RM15, RM22	N
RM18	21	"	(A+(N+1))>=abs(B)		RM16, RM21	N
RM19	21	"	(A+N)>=abs(B-1)			Y
RM20	21	"	(A+N)>=abs(B+1)			Y
RM21	21	"	(A+N)>=(abs(B)-1)		RM16, RM18	N
RM22	21	"	(A+N)>=(abs(B)+1)		RM15, RM17	N
RM23	27	(B>0)	(B-1)>0	(B>1)		Y
RM24	27	"	(B+1)>0	(B>-1)	Original	N
RM25	29	(abs(A+N)>=B)	abs((A-1)+N)>=B		RM27	Y
RM26	29	"	abs((A+1)+N)>=B		RM28	Y
RM27	29	"	abs(A+(N-1))>=B		RM25	N
RM28	29	"	abs(A+(N+1))>=B		RM26	N
RM29	29	"	(abs(A+N)-1)>=B		RM32	Y
RM30	29	"	(abs(A+N)+1)>=B		RM31	Y
RM31	29	"	abs(A+N)>=(B-1)		RM30	N
RM32	29	"	abs(A+N)>=(B+1)		RM29	N
RM33	35	((A-N)<=B)	((A-1)-N)<=B		RM36, RM38	Y
RM34	35	"	((A+1)-N)<=B		RM35, RM37	Y
RM35	35	"	(A-(N-1))<=B		RM34, RM37	N
RM36	35	"	(A-(N+1))<=B		RM33, RM38	N
RM37	35	"	(A-N)<=(B-1)		RM34, RM35	N
RM38	35	"	(A-N)<=(B+1)		RM33, RM36	N

Table 7.9 - Mutants produced from the REMAINDER procedure using the mutation operator, predicate alteration by a small amount

Of these 38 mutants, 29 are classified as identical. Of the 29 identical mutants, 26 are identical with some other mutant produced and 3 are identical (or equivalent) with the original REMAINDER procedure. Mutants 6, 8 and 24 are equivalent since the values of A and B required to reveal them would not take control to the mutated statement. Control is deviated above the mutation, hence the necessary output values cannot be produced. Of the 38 mutants produced, 19 are to be used for mutation analysis of the REMAINDER.

For each of the 19 runs, a single, randomly generated initial point is used. The random values are generated in the range ± 100 for each input variable. The node iteration threshold is 500 and the input variables are considered in the order A first, then B.

To speed up and simplify the mutation testing process, output from a mutant is not compared with output from the original procedure. A substitute for the original is used; the intrinsic Ada "rem", remainder function. When the mutant's output is produced the Ada "rem" function is called with the same point used on the mutant. The mutant's output and the Ada "rem" function's output is compared. If there is a difference, it is reported.

7.5.2 Mutation Analysis Results and Discussion

Table 7.10 shows the results obtained.

Mutant	Initial point		Result	If Revealed :		
	A	B		Heuristic used	Revealing point	
					A	B
RM1	-31	93	Difference	DA	1	93
RM2	91	21	Difference	DA	-1	21
RM3	9	-98	Stuck in loop	DA	9	0
RM4	-77	-14	Stuck in loop	DA	-77	0
RM5	88	-68	Stuck in loop	LP	1	1
RM7	69	-34	Stuck in loop	DA	69	1
RM9	-29	18	Difference	DA	1	1
RM10	-49	-59	Difference	DA	1	1
RM15	80	52	Difference	DA	80	-1
RM16	82	84	Difference	DA	82	-1
RM19	76	4	Difference	DA	76	-1
RM20	-85	72	Difference	DA	1	-1
RM23	-61	61	Stuck in loop	DA	-61	1
RM25	-79	-57	Difference	DA	-79	1
RM26	-36	100	Difference	DA	-36	1
RM29	-37	-81	Difference	DA	-37	1
RM30	-52	67	Stuck in loop	DA	-52	1
RM33	-31	-52	Difference	DA	-31	-1
RM34	72	-10	Difference	DA	-1	-1

Table 7.10 - Results from mutation testing the REMAINDER procedure

In 6 of the runs, control became stuck in one of the REMAINDER's four loops. Consequently, there is no output and nothing to compare with the Ada intrinsic remainder function. However, we can assume, and this is supported by Abbot (1986), that if a mutant does not terminate, it can be considered revealed.

One hundred percent mutation adequacy is achieved using one HATS run for each initial point. The DA revealed 18 of the 19 mutants (95%). The DA achieves this through generating boundary located points for some of the 10 predicates it can be applied to. These predicates control nodes 2 to 10 in figure 7.2. Boundary located points reveal mutants produced by the mutation operator, predicate alteration by a small amount.

Considering the original condition controlling nodes 6 and 7 ($A > 0$) and its mutation ($A > 1$), it is apparent that variable A requires a value of 1 for the mutant to be revealed. In mutant RM5's run, the DA generated the point (1, -68) on node 3, but no difference in output was reported. Instrumentation in the mutant revealed that an incorrect path had been taken but the correct output was produced. Later, on node 19, the LP generated the point (1, 1) which revealed mutant RM5.

7.6 The LINEAR_SEARCH Procedure

The LINEAR_SEARCH (figure 7.9) performs a sequential search for a given integer value on a list of integer values. If the given value is located, its location in the list is returned, otherwise, the value 0 is returned. The procedure's control flow tree is shown in figure 7.10.

```
Control flow tree node number
   Line number
      Ada statements

    1  procedure LINEAR_SEARCH
         ( A in INTARRAY; X in INTEGER; Y out INTEGER ) is
    2    l : POSITIVE;
    3  begin
1[  4    I := 1;
 L  5    while ( I <= A'LAST ) and then ( A(I) /= X ) loop
2[  6      I := l + 1;
    7    end loop;
    8    if ( I <= A'LAST ) then
    9      Y := l;
   10    else
   11      Y := 0;
   12    end if;
   13  end LINEAR_SEARCH;
```

Figure 7.9 - The LINEAR_SEARCH procedure

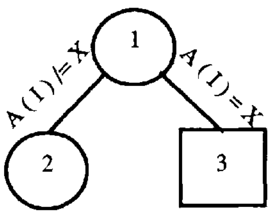


Figure 7.10 - Control flow tree of the LINEAR_SEARCH procedure

The LINEAR_SEARCH procedure has 2 input variables. Variable A, is a five element integer array and contains the values to be searched. Integer variable X contains the value to be searched for. The only output variable is integer variable Y which stores the index of X where Y is located, or 0 otherwise.

The Ada code (figure 7.9) has two conditional statements, however, only one conditional statement is modelled in the control flow tree (figure 7.10). This is due to the "if" statement condition at line 8 having no direct data flow influence from the input variables. Hence, the branches it controls cannot be affected by a heuristic.

These branches are controlled by the number of iterations the while loop takes, which is affected by a heuristic. Consequently, this "if" statement is not represented in the control flow tree.

7.7 Techniques for Testing Arrays

The techniques used are the same as those used on scalar variables. Each element is treat the same as a scalar variable. Since HATS is a dynamic testing system it does not suffer from the array reference difficulties that symbolic execution has (Coward, 1988). Instrumentation has access to actual variable values. By monitoring a predicate value, a heuristic can determine which element of an array influences the considered predicate. The instrumentation for a scalar variable condition and an array variable condition are compared in figure 7.11.

```
if ( A=B ) then
 PV = ( A-B );
 UPDATE_LEFT_NODE (..,PV,..);
 : :
else
 PV = ( A-B );
 UPDATE_RIGHT_NODE (..,PV,..);
 : :
end if;
```

Where A and B are integer input variables

Instrumentation for a Condition Involving Integer Variables

```
if ( A(I)=B ) then
 PV = ( A(I)-B );
 UPDATE_LEFT_NODE (..,PV,..);
 : :
else
 PV = ( A(I)-B );
 UPDATE_RIGHT_NODE (..,PV,..);
 : :
end if;
```

Where A is an integer array input variable and, B and I are integer input variables

Instrumentation for a Condition Involving an Integer Array and an Integer Variable

Figure 7.11 - Comparison of instrumentation for scalar variable and array variable conditions

The LINEAR_SEARCH has a loop that is controlled by an equality operator between an array variable, A, and a scalar variable, X. To satisfy a loop's branches, control must enter the loop on one run and on another run, must not enter the loop. This corresponds to 1 or more loop iterations and 0 loop iterations. Therefore branch satisfaction is achieved through the values stored in the first element of array A and variable X. The remaining elements in the array have no influence over branch coverage for the loop concerned. However, they could affect additional statements placed inside the loop and statements before and after the loop.

7.8 LINEAR_SEARCH Branch Testing Experiments

7.8.1 HATS Experimental Set Up

Five initial points have been hand selected in the range ±100 for each input variable. A concern with automatic test data generation is its management of arrays, since effort may be wasted on non-influential input variables. To illustrate how HATS fairs, the order input variables are considered in has been modified. The initial points and the order input variables are to be considered in is shown in table 7.11. The previous section (7.7) pinpointed the influential variables X and A(1).

Run	Input variables and values					
	Considered first				Considered last	
LS1	X	A(1)	A(2)	A(3)	A(4)	A(5)
point	12	1	4	3	5	3
LS2	X	A(1)	A(2)	A(3)	A(4)	A(5)
point	41	-53	-35	-37	91	92
LS3	A(1)	A(2)	A(3)	A(4)	A(5)	X
point	41	-53	-35	-37	91	92
LS4	A(5)	A(4)	A(3)	A(2)	A(1)	X
point	41	-43	82	-21	-93	27

Table 7.11 - LINEAR_SEARCH's initial points and the order heuristics will consider input variables

The two influential variables will be the first two considered by heuristics in runs LS1 and LS2. In run LS3, A(1) will be considered first and X, last. In run LS4, the two variables will be considered last. The node iteration threshold is 500.

7.8.2 Run LS1

The initial point traversed the path 1, 2 (figure 7.10) and had a node 2 predicate value of -11. On node 3, the LP found a solution point after 6 iterations (table 7.12). There were no problems over identifying which input variables were influential. Only variable X was modified, all other input variables remained constant.

Iter	Phase	Generate value X	Execute value X	Trav effect	Pred value	Next action
1	DL	12	13	ST	-12	-X
2	DL	12	11	ST	-10	+X
3	DL	12	14	ST	-13	+X
4	DL	12	15	ST	-14	PP
5	PP	1	2	ST	-1	PP
6	PP	2	1	NT	0	SUCC

Table 7.12 - Run LS1 excerpt : LP considering node 3, modifying variable X; array A stayed constant with elements 1 to 5 having values 1, 4, 3, 5 and 3 respectively

7.8.3 Run LS2

The order input variables were considered is the same as the previous run, LS1. The initial point traversed the path 1, 2. On node 3, the LP modified the first variable X. A solution point was located at (-53, -53, -35, -37, 91, 92) for variables X and A(1..5) respectively. The LP took 6 iterations.

7.8.4 Run LS3

Although the same point is used as run LS2, the allocation of values to variables has changed and so has the order variables are considered in. The initial point traversed the path 1, 2. On node 3, the LP modified the first element of array A. A solution point was located at (92, -53, -35, -37, 91, 92), for variables A (1..5) and X respectively, after 6 iterations.

7.8.5 Run LS4

This run considers the hardest scenario for HATS, where the influential variables, A(1) and X, are not considered until last. The initial point traversed the path 1, 2, producing a predicate value of -120. On node 3, the LP first considered variable A(5). Table 7.13 shows there is no change in the predicate value. This enables the LP to detect a non-influential input variable. The same occurred with the next three variables considered, A(4), A(3) and A(2).

Iter	Generate value	Execute value	Trav effect	Pred value	Next action
1	41	42	ST	-120	-A(5)
2	41	40	ST	-120	+A(5)
3	41	43	ST	-120	+A(5)
4	41	44	ST	-120	TNV

Table 7.13 - Run LS4 excerpt : LP considering node 3, modifying variable A(5) in the Determine-linearity phase; array A (1..4) and X stayed constant having values -93, -21, 82, -43 and 27 respectively

Iter	Phase	Generate value A(1)	Execute value A(1)	Trav effect	Pred value	Next action
17	DL	-93	-92	ST	-119	-A(1)
18	DL	-93	-94	ST	-121	+A(1)
19	DL	-93	-91	ST	-118	+A(1)
20	DL	-93	-90	ST	-117	PP
21	PP	27	26	ST	-1	PP
22	PP	26	27	NT	0	SUCC

Table 7.14 - Run LS4 excerpt : LP considering node 3, modifying variable A(1); array A (2..5) and X stayed constant having values -21, 82, -43, 41 and 27 respectively

After 16 iterations the LP considered the first influential variable A(1). Modifying this variable, the LP located a solution point at (27, 27, -21, 82, -43, 41) for variables X, A (1..5) respectively, after 22 iterations (table 7.14).

7.8.6 HATS Discussion

The LP ensured that complete branch coverage was achieved in every run. There was no impact from upper-deviations since they could not occur. A concern with arrays as input variables is the ability to identify which elements are influential in the considered predicate. The LP took only 4 iterations over a non-influential input variable and concentrated on influential input variables when they were located.

7.8.7 Comparison of HATS with Random Testing on the LINEAR_SEARCH Procedure

The random testing range for each input variable is ± 100, which is the same as HATS's initial point selection range. Each random testing run continues until all nodes are traversed. The number of iterations taken to traverse each node is recorded. The average iterations per node, over 500 runs, is calculated and used. Results for HATS are taken from runs LS1 to LS4.

Node	HATS iterations	Random iterations
1	IPT	1
2	IPT	1.002
3	6-22	210.776

Key :	
IPT	Initial point traversed
CT	Coincidentally traversed

Table 7.15 - The number of iterations taken by HATS and random testing for each of the LINEAR_SEARCH's nodes

Node 2 has a very large solution domain. This is shown in table 7.15 by the number of iterations random testing takes. On node 3, HATS makes a good improvement over random.

8 Conclusions and Further Work

8.1 Introduction

This chapter draws conclusions from the research presented in this thesis. The limitations of the approach presented are discussed, and some suggestions on how they may be overcome and how the approach may be extended are given.

8.2 Conclusions

8.2.1 HATS is an Improvement Over Random Testing

The random test data generation effort (number of points generated) required to traverse a node is dependent upon the size and form of the node's solution domain. The effort HATS must expend to traverse a node is dependent upon the size and form of the node's solution domain, the form of the partial path function to the node and the number of branches in the partial path to the node.

The following properties have been identified to be present when HATS takes less effort than random testing.

- When a considered node's domain occupies, on average, 9% or less of the test software's input space.
- When the considered node's partial path function is linear.
- When the value to satisfy the considered node's predicate can be determined.
- When a node's domain is very small and has frequently located points along a single boundary.
- When a solution point is local to the current search and can be located in a co-ordinate direction.

The above list requires explanation. The considered node's domain size as a percentage of the test software's input space, where HATS will typically take less iterations then random testing, is calculated as follows. From all non-mutation runs, the size of the considered nodes' domains, as a percentage of input space, where HATS took less iterations than random to traverse, is averaged. On some nodes the difference in effort can be marginal and others, considerable. In practice, the node's domain size can be much larger; up to 48%. This is due to initial point traversal or coincidental traversal of the node. Also in practice, the node's domain size can be considerably smaller than 9%. This has only occurred when the node's domain is located on a notional boundary and HATS has failed to locate a solution before the node iteration threshold is reached. Random generation is able to locate a solution, but using a very large number of iterations.

Due to the nature of software, the items in the above list cannot be regarded in isolation; there is a relationship between them. Determining when HATS should be an improvement over random could involve one or more of the items. These observations have been made through the use of the four heuristics, DA, AV, LP and BF. Should other heuristics be used then the properties that help HATS to improve on random testing may change.

8.2.2 Domain Boundaries Can be Used as a Guide to Solution Points

When solution points to the considered node are sparsely located on a domain boundary, the boundary can be used as a guide to them. The Boundary Follower heuristic has demonstrated this. A spin-off from boundary following is that only boundary located points are generated (boundary value test data).

8.2.3 Control Deviations From Partial Paths to a Considered Node Have a Detrimental Affect on Heuristic Performance

When a heuristic's operation depends on the comparison of predicate values, control deviations from the one or more partial paths to a considered node have a detrimental affect on the heuristic's performance. This affect is not to be under-estimated; it can be significant and cause a heuristic to fail; in occasional cases, even when a solution point is very close. When a control deviation from the partial path to the considered node occurs the heuristic is aware of this but there is no predicate value to compare with previous predicate values. Hence, the heuristic must take some appropriate action to cause control to resume execution through the considered node's parent node and produce a predicate value. This action upsets the normal flow of the heuristic and uses iterations.

The potential for upper-deviations is related to the number of branches above the considered node and the partial paths' function.

8.2.4 Termination Criteria Based on Promising Effects are Effective and Efficient

It is important that a heuristic's termination criteria are effective otherwise many iterations can be wasted or a search cut short before a solution is found. The Linear Predictor's termination criteria, which are based on promising effects, have been shown to be effective and efficient.

8.2.5 Coincidental Traversal Can be Considerable

Coincidental traversal occurs in two cases. First, when control is following a path already traversed and deviates at some branch, traversing a previously untraversed node. If the considered node has successor nodes that are untraversed, control may traverse these. Second, when a considered node is traversed and control continues to traverse previously untraversed nodes that are successors of the considered node. The two cases are related through the traversal of previously untraversed successor nodes.

Coincidentally traversed nodes tend to be the nodes that would only require a small amount of heuristic effort to traverse; these are easy nodes. The number of coincidentally traversed nodes in a run is dependent on the path taken by the initial point. If this leaves a number of easy nodes then there is a higher chance of coincidental traversal taking place.

The number of coincidentally traversed nodes can be considerable. The maximum achieved in the runs presented in this thesis was 47% in run R2. The average coincidental coverage from all non-mutant runs is 29%. TRIANGLE_COMPLETE and REMAINDER had a higher coincidental coverage than the other procedures, due to them having more nodes than the other procedures tested.

Generally, on runs where the initial point differed but the same initial path was traversed, the same nodes are coincidentally traversed, since generally the same nodes are considered.

8.2.6 The Initial Point Set Influences the Effort and Success of the Heuristics

Since the heuristics are localised searches the initial point(s) for a run influence the number of iterations a heuristic takes and its success. This influence may be direct when a heuristic is using an initial point or indirect when a heuristic is using a point descended from an initial point. It is particularly evident when a considered node's solution domain is small or there is a high potential for upper-deviation.

8.2.7 Point Metrics Based on the Closeness Phenomenon have Limitations

The point closeness metric has led heuristics to a location in the input space where the metric indicated a solution would be, however, a solution is not always there. When the solution domain is small and sparsely located, this increases the chance of a heuristic being mis-led. The point closeness metric often leads a heuristic to a notional boundary where there is no solution. The responsibility of identifying and

managing this situation lies with the heuristic. Presently, this responsibility is not executed very well due to the localised, non-random nature of the heuristics.
Cross, et al, (1991) refer to the closeness phenomenon as "goodness". However, the closest point will not necessarily take a heuristic to a solution point, since the metric does not take into consideration the structure of the input space the heuristic is moving into, particularly where domain boundaries may be crossed before a solution is found. Nevertheless, the metric is good for many of the predicates considered.

8.2.8 Heuristic Based Test Data Generation is a Promising Approach

Software is composed of many diverse functions. To find specific points that cause a required effect in the software is more-than-likely beyond the capability of a single algorithm. HATS has addressed this through the ability to contain a number of diverse heuristics that are selected from a hierarchy for the current problem.
HATS is an improvement over random testing and does not suffer from the intrinsic problems other approaches do. However, the present heuristic approach has limitations, which are outlined with potential remedies in section 8.3.
The heuristic approach benefits from coincidental traversal and the fortuitous co-operation of heuristics, where points generated by one heuristic are used to success by another heuristic.

8.3 Further Work

8.3.1 HATS

8.3.1.1 Automation of Processes the HATS Harness Depends on

Only test data generation has been automated in HATS. Processes such as the generation of the test software model and the instrumentation of the test software are undertaken manually. However, their automation is well understood (Yau and Grabow, 1981; Cimittle and Carlini, 1991) and could be implemented in future. Once in place they would speed up the whole testing process and aid the use of different test criteria and software models.

8.3.1.2 Heuristic Selection and Termination

Heuristic selection aims to choose a good heuristic first. A static hierarchy favours the general heuristics rather than the more specific heuristic, which if required to solve the problem, will not be used until some effort has been expended. Each of the heuristics should be described to HATS in terms of exploitable test software characteristics it can use to influence control toward the consid the consi(element (i.e. branch). This description can be used to determin) determi suitable. HATS could determine test software characteristics fi :teristics : HATS would then produce a selection hierarchy from comparii 1 compar descriptions against the determinable characteristics of the test)f the test hierarchy may be updated as further data is produced from the from the To ensure that a heuristic does not get stuck or take overly long verly lon software element, a threshold is required. A static threshold (n reshold (1 be sufficiently large to ensure that a feasible considered softwa: ed softw: traversed at the cost of wasting iterations on an infeasible or to(sible or tc Furthermore, a promising heuristic can be left short of iteration f iteratio used on unsuccessful heuristics. As an alternative to static, test static, tes based thresholds (i.e. node iteration) a threshold for each heuris ach heuri determined from the characteristics used in heuristic selection. selection. heuristic a chance. A limit on the number of heuristics that co(:s that co considered software element would need to be determined. iined.

8.3.1.3 Determining Influential Input Variables

Modifying only input variables that are influential in the consid the consi is an important issue. When there are many input variables, mı iables, m wasted modifying non-influential input variables. Korel (1990:)rel (199(dynamic data flow analysis to rank input variables according to cording t(equal to the number of predicates in the partial path to the cons:) the con: element, where the input variable influences control flow on th(low on tl different approach revealed through the experiments conducted :onducte(predicate value analysis. Each input variable of a point that ca1 nt that ca traversal to the considered element is modified by an equal-sm: equal-sm in the considered elements predicate value would determine wh :rmine w considered predicate influence and to what extent. A comparis(compari: approaches would be beneficial, particularly to determine dynar nine dyn: dependence on the partial path used and predicate value analysi: ue analys point used.

8.3.1.4 Guiding Control to the Considered Software Elemei re Elem(

HATS does not manage upper-deviations where they occur. A occur. A remedy the problem from the considered node. Consequently, tequently, localising the search. Korel (1992) suggests that the search nee(search ne with critical branches that do not allow execution of a specific software element. Hence, the search would be focused on interfering predicates. However, this could adversely affect boundary exploiting heuristics. A comparison between these two techniques and others would be valuable. In general, the form of the input space surrounding the point causing the upper-deviation may determine the most suitable technique.

8.3.2 Heuristics

8.3.2.1 Limitations of the Existing Heuristics

In general the existing heuristics are limited through being a localised, co-ordinate-direction search. Improvements for the existing heuristics are outlined, however, I believe that the most promising results may be gained from exploring heuristics from classes other than localised, co-ordinate-direction search.

8.3.2.2 Improvements to Existing Heuristics

Direct Assignment

To handle more complex predicate expressions, further rules could be implemented. Rules proposed by Howden (1987) could be used as a basis. Deason, et al's, (1991) system is based on Howden's rules. Clearly this would involve further static analysis of the test software and storage of the results.

Alternating Variable and Linear Predictor

These two heuristics are similar in the way they modify input variables. However, they are complementary; the AV is effective with non-linear partial path functions and conversely, the LP is effective with linear partial path functions. Hence, for improvement these two heuristics could be merged into one. The new heuristic would use an updated form of the LP's termination criteria which would support the AV's method.

Boundary Follower

The BF has several limitations; it is constrained to operate in two dimensions, it only follows a boundary in one direction, it has no termination criteria and it has difficulty when there is more than one boundary. Addressing the dimensionality of the search and operating in the presence of many boundaries, a moving window in input space should be investigated. Presently, the BF has only a single point to view the input space with and make operational decisions from. The window, formed of the predicate values over a range of values for the input variables being modified, would give the BF an improved input space view. The window may be formed of the perimeter points or be solid, and may change size and dimension. A modified form of Nelder and Mead's (1965) Simplex Search may be suitable.

Addressing boundary movement direction, a possibility is to use a metric that indicates the potential success of the present search along a boundary in one direction. This would be combined with a backtracking technique that would reposition the heuristic at some other input space point, should the present search prove to be fruitless. Addressing termination criteria, a further metric could be based on the heuristic's backtracking, or more simply, a threshold.

8.3.2.3 The Search for Better Heuristics

The heuristic approach relies on many varied heuristics which can tackle the many problems that test data generation presents. Developing or locating, and evaluating heuristics should be an on-going process. There is a wealth of existing, modern optimisation techniques which have not been applied to test data generation (Conn, et al, 1994). Many of the first generation optimisation techniques have yet to be applied (Murray, 1972; Gill and Murray, 1974). Although these references span several classes of techniques, direct search still appears to be the most suitable, because it makes few assumptions about the optimised function. Deason, et al, (1991) have concentrated on simpler heuristic techniques which could be built upon.

Some recently developed global optimisation techniques which can operate in the presence of constraints show particular promise (Goldberg, 1989; Rabinowitz, 1995). Regardless of which heuristics are used, there is a need to clearly understand the characteristics of software from an adaptive test data generation point of view. This would help in many ways; test software could be better defined as an optimisation problem and suitable optimisation techniques could be selected; optimisation techniques could be adapted to perform better on the test software.

8.3.3 Alternative Representations of the Test Software as a Test Data Generation Problem

The test software may be represented or viewed in different ways as a test data generation problem for the heuristics to solve. The representation potentially involves several aspects; some of which follow.

Software Model

The choice of software model may have significant implications. HATS presently uses a control flow tree to focus testing attention and store data at suitable locations. Other models may offer greater versatility (Cimittle and Carlini, 1991; Yau and Grabow, 1981).
Influencing control flow to the considered element in the test software has been identified as a difficulty. The structural path-prefix approach depends on control flow passing close to the considered test software element. A number of approaches exist to manage this. The traversal of a unique partial path can be required (Korel, 1992) with back tracking to manage control deviations. HATS adopts a more relaxed approach where any partial path to the considered element may be traversed. Control deviations are explicitly recognised and the heuristics attempt to influence control flow back to a partial path bringing control flow close to the considered element. Korel (1992) later outlines a modification where the traversal of a unique partial path is relaxed and any partial path that leads to the considered element is acceptable. Alternative test point metrics may remove the need for a heuristic to explicitly manage the control deviation problem. New metrics could be based on penalty functions (Gill and Murray, 1974) which transform the constrained optimisation problem, which test software is, into an unconstrained problem. This new metric would be combined with a variable partial path to the considered software element. Gallagher and Narasimhan (1993) have used penalty functions.

Test Criteria

There are alternative ways the test criteria could be managed toward its satisfaction. Many approaches, including HATS, consider each software element in turn. Other approaches (Roper, et al, 1995) consider the overall test criterion satisfaction level.

A few representations and alternatives have been outlined above. Further research may reveal other representations. Better understanding software as a test data generation problem is vital so that the most suitable test software representation can be used for the required testing criteria. Clearly, comparing the above representations with various test software and test criteria would be a valuable exercise.

8.3.4 Metrics

Metrics play an important part in adaptive test data generation. Most metrics used in adaptive test data generators are based on the closeness phenomenon. However, this has been shown to have limited ability. Clearly, more accurate metrics could have significant beneficial impact on a heuristic's success. The development of metrics to help heuristics in other ways is an interesting area. Two areas that could benefit are determining a heuristic's suitability to solve a given problem and determining the potential success or failure of a heuristic subsequent to it running for some time.

8.3.5 Scope of Testable Software

Automatic test data generators still have some way to go before they can be used productively on industrial software without some effort to help them. Increasing the diversity and complexity of software that useful test data can be generated for, is important. This includes testing further statements, data types and other features, such as a languages support for objects. Increasing the complexity of software that can be tested is also important.

Testing further data types presents a problem with respect to the point metrics and the heuristics. Both must be aware of the data type and handle them accordingly. This places an additional level of complexity on both. The closeness phenomenon does not exist for some data types such as pointers. Genetic Algorithms (Goldberg, 1989) overcome this issue by manipulating data types as a bit stream rather than through the operations each data type allows.

Increasing the complexity of testable conditions, to compound conditions, involves the point metric. One approach is to assign weights to each simple condition so that when the compound condition is true it has a value that is produced from each of the simple condition's values and is above some threshold. This would only support branch testing. To support multiple-condition testing, a truth table, representing each simple condition, could be associated with the compound condition. The heuristic's objective would be to complete the truth table to some degree.

8.3.6 Input Space Study

The input space for software of only a few lines with loops, can be quite complex. Larger software with many input variables has more-than-likely a very complex input space. Nevertheless, the heuristics must navigate through the input space in search of a solution point. Studying the input space of a large sample of software would reveal useful information that would help in many ways. It would help the development and improvement of heuristics and metrics. It would aid a comparison with numerical optimisation problems and help in the selection of suitable optimisation techniques. White and Cohen (1980) have conducted significant research in this area.

8.3.7 New Tools

During the course of this research, tools that would help researchers of automatic test data generation and possibly other areas, have become apparent.

8.3.7.1 Input Space Navigator

This would enable the operator to view a 2 dimensional window of predicate values, or some other metric's values, for some location in the test software's input space. The window is produced by executing the points from a range of values for two of the input variables. Controls would enable the input space window to be moved, the non-displayed dimensions to be adjusted and the displayed dimensions to be exchanged with the non-displayed dimensions. This tool automates the partial input planes seen in this thesis. The tool would be particularly helpful when analysing the points selected by a heuristic and looking at areas of the input space where the heuristic is having difficulty. It would be possible to link HATS to this tool so that heuristic's input space navigation may be shown in real time. Alternatively the tool could retrieve test data from storage and replay the heuristics navigation after the heuristic has executed.

8.3.7.2 Path-based Variable Mapping Viewer

This tool graphically displays the mappings between the values (or ranges of values) of variables for the path taken by some point. Each variable's range would be shown as a line representing all the possible values the variable could assume. When a variable's value is used in the definition of another variable, a line would be drawn from the used variable's value, in its range, to the defined variable's new value, in its range. Of course, there can be many variables used in the definition of another variable. The input variables' ranges would be shown first and their values could be adjusted. The length of the mapping network would be proportional to the length of the data flow path. Should some point cause the same data flow path to be taken as the previous point then only a slight change in mappings should be observed. Should some point cause a different data flow path to be taken to the previous point then the variables and their order in the mapping network would change. This tool would be useful to see if, and to what extent, an input variable influences other variables.

Appendix 1 HATS User Summary

The steps involved in using the Heuristically Aided Testing System (HATS) are outlined in figure A1.1.

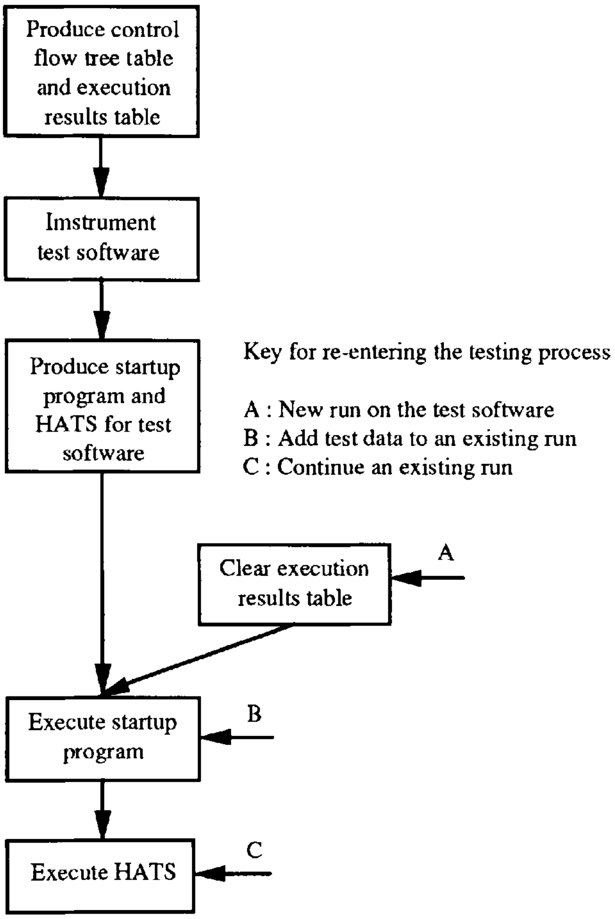

Produce control flow tree table and execution results table
Instrument test software
Produce startup program and HATS for test software
Key for re-entering the testing process
A : New run on the test software
B : Add test data to an existing run
C : Continue an existing run
Clear execution results table
A
Execute startup program
B
Execute HATS
C

Figure A1.1 - Flow diagram showing the HATS testing process

The first step in the HATS testing process is to produce two tables in an Oracle database. The control flow tree table, which must be produced manually, stores a representation of the test software's control flow tree. Each record holds information on each node in the tree. The execution results table is where data generated from the startup program and HATS are stored. A record in this table represents the execution of a node in the control flow tree. The second step in the HATS testing process is to instrument the test software and this must be done by hand. Instrumentation is placed at the following places in the test software: the beginning, after each branch and at the end.

The next step is to produce the startup program and HATS for the test software being used. The startup program enables test points to be entered and the test software to be executed upon them. This step is achieved by specifying the data types and number of input variables, then compiling the startup program and HATS with the instrumented version of the test software.

The next step, clearing the execution results table is only necessary if the testing process is being re-entered. On the first time through the table will already be clear as it has just been created. Clearing this table is achieved by deleting all the rows from the execution results table. Re-entering the HATS testing process at this point enables new runs on the test software to be made and is denoted by label A in figure A1.1. A test software *run* consists of one or more iterations where the execution results table is not cleared. An *iteration* is the execution of a single test point on the test software.

The next step is to execute the startup program. It is necessary to run this program before HATS since it executes the test software and provides a starting point for HATS to work from. Each time the startup program is run it accepts a test point and then executes the in-built test software. The startup program can be run any number of times. Re-entering the HATS testing process at this point, depicted as label B in figure A1.1, enables the user to add further test points to an existing HATS run.

The final step in the HATS testing process is to execute HATS. HATS attempts to consider each untraversed node in the control flow tree. Its objective is to generate new test points from existing test points in the control flow tree that will traverse the, as yet, untraversed node, under consideration. It may not be possible to consider all untraversed nodes as some may be infeasible and others too difficult for the heuristics to solve. Re-entering the HATS testing process at this point, shown by label C in figure A1.1, enables a run to be continued after a break in execution.

Appendix 2 Quadratic Mutation Testing Initial Points

Table A2.1 contains the initial points chosen for the first round of mutation analysis on the QUADRATIC procedure. Each mutant run has an identifier, which is designated by the letters QM followed by a unique number, eg QM1.

Mutant	Initial point		
	A	B	C
QM1	227	262	26
QM2	253	798	746
QM3	153	76	478
QM4	208	611	352
QM5	343	190	561
QM6	561	749	414
QM7	63	327	889
QM8	756	752	654
QM9	458	154	291
QM10	143	845	369
QM11	761	291	768
QM12	80	873	264
QM13	801	319	363
QM14	713	111	70
QM15	105	770	969
QM16	421	821	334
QM17	104	814	289
QM18	252	469	652
QM19	654	109	554
QM20	355	868	992
QM22	252	608	493
QM23	29	578	3
QM24	60	338	573
QM25	631	325	997
QM26	71	781	920
QM27	700	540	706
QM28	428	492	476
QM29	88	522	408
QM30	973	979	361
QM31	322	31	586
QM32	282	436	265
QM33	427	188	608
QM35	885	331	257
QM36	99	591	888
QM40	249	763	713
QM41	941	95	979
QM42	432	294	753
QM43	51	96	905
QM44	602	663	792
QM45	6	779	98
QM46	808	939	202
QM47	337	907	168
QM48	743	884	589
QM49	322	517	102
QM50	981	462	293
QM51	801	478	544
QM52	187	919	890
QM53	300	232	903
QM54	5	760	649

Table A2.1 - Initial points for the first round of QUADRATIC mutation analysis

Appendix 3 Heuristics' Pseudo Code

A3.1 Direct Assignment Heuristic

A3.1.1 First Iteration Set-up

```
DA_setup
   consid_inp_var := input variable in considered predicate
   case ( considered predicate's relational operator ) is
   when '<' =>
      instructions := decrease by 1
   when '>' =>
      instructions := increase by 1
   when '/=' =>
      instructions := increase by 1
   when '=' | '<=' | '>=' =>
      instructions := no change
   end case
   generate_point := first point to traverse sibling node
         with considered predicate's constant for consid_inp_var
end DA_setup
```

A3.1.2 Generator

```
DA_generator
   case ( modification direction of instructions ) is
   when increase =>
      execute_point := add instructions value to consid_inp_var of
            generate_point
   when decrease =>
      execute_point := subtract instructions value from
            consid_inp_var of generate_point
   when no change =>
      execute_point := generate_point
   end case
end DA_generator
```

A3.1.3 Duplicate Data Handler

```
DA_duplicate_data_handler
   terminate HATS harness
end DA_duplicate_data_handler
```

A3.1.4 Evaluator

```
DA_evaluator
   if ( considered node traversed ) then
      heuristic is successful
   else
      add considered node and subtree to unreachable nodes list
      heuristic has failed
   end if
end DA_evaluator
```

A3.2 Alternating Variable Heuristic

A3.2.1 First Iteration Set-up

```
AV_exploratory_phase_setup
   phase := exploratory
   consid_inp_var := first input variable
   determine_exploratory_instructions
end AV_exploratory_phase_setup

determine_exploratory_instructions
   if ( only 1 point has traversed the sibling node ) then
      instructions := increase by 1
      generate_point := first sibling point
   else
      case ( comparison of the considered input variable of the
         two closest sibling_traversal points ) is
      when closest point var less than the next closest test
            point var =>
         if ( required predicate value is inbetween the two
            closest predicate values ) then
            instructions := increase by 1
         else
            instructions := decrease by 1
         end if
      when closest point var equal to next closest point
            var =>
         instructions := increase by 1
      when closest point var greater than the next closest
            point var =>
         if ( required predicate value is inbetween the two
            closest predicate values ) then
            instructions := decrease by 1
         else
            instructions := increase by 1
         end if
      end case
      generate_point := the closest sibling_traversal point
   end if
end determine_exploratory_instructions
```

A3.2.2 Generator

```
AV_generator
   case ( modification direction of instructions ) is
   when increase =>
      execute_point := add instructions value to consid_inp_var of
         generate_point
   when decrease =>
      execute_point := subtract instructions value from
         consid_inp_var of generate_point
   end case
end AV_generator
```

A3.2.3 Duplicate Data Handler

```
AV_duplicate_data_handler
   case ( modification direction of instructions ) is
   when increase =>
      add 1 to consid_inp_var of generate_point
   when decrease =>
      subtract 1 from consid_inp_var of generate_point
   end case
end AV_duplicate_data_handler
```

A3.2.4 Evaluator

```
AV_evaluator
   case ( traversal effect of execute_point) is
   when node_traversal =>
      heuristic is successful
   when sibling_traversal =>
      case ( closeness of execute_point to expected boundary
         compared to closest point stored ) is
      when closer to expected boundary =>
         closer_sibling_traversal_evaluator
      when further from expected boundary =>
         further_sibling_traversal_evaluator
      end case
   when upper_deviation =>
      upper_deviation_evaluator
   end case
end AV_evaluator

closer_sibling_traversal_evaluator
   case ( phase ) is
   when exploratory =>
      phase := pattern
      increase_pattern_step
   when pattern =>
      if ( required predicate value is in between the two closest
            predicate values ) then
         reverse_modification_direction
         much_reduce_pattern_step
      else
         increase_pattern_step
      end if
   end case
   generate_point := execute_point
end closer_sibling_traversal_evaluator

further_sibling_traversal_evaluator
   if ( further sibling traversal threshold exceeded ) then
      try_next_var
   else
      case ( phase ) is
      when exploratory =>
         if ( this is the first further sibling_traversal on the
               considered input variable ) then
            reverse_modification_direction
            generate_point := the closest point stored
         else
            try_next_var
         end if
      when pattern =>
         if ( pattern further sibling_traversal in succession
               threshold exceeded ) then
            try_next_var
         else
            much_reduce_pattern_step
            generate_point := the closest point stored
         end if
      end case
   end if
end further_sibling_traversal_evaluator
```

```
upper_deviation_evaluator
   if ( upper_deviation threshold exceeded ) then
      try_next_var
   else
      case ( phase ) is
      when exploratory =>
         if ( this is the first upper_deviation on the considered
               input variable ) then
            reverse_modification_direction
            generate_point := the closest point stored
         else
            try_next_var
         end if
      when pattern =>
         phase := exploratory
         instructions value := 1
         generate_point := the closest point stored
      end case
   end if
end upper_deviation_evaluator

increase_pattern_step
   instructions value := instructions value * 2
end increase_pattern_step

reverse_modification_direction
   case ( modification direction in instructions ) is
   when increase =>
      instructions direction := decrease
   when decrease =>
      instructions direction := increase
end reverse_modification_direction

much_reduce_pattern_step
   instructions value := instructions value / 5;
   if ( instructions value < 1 ) then
      instructions value := 1;
   end if
end much_reduce_pattern_step
```

A3.2.4.1 Abandoning Consideration of an Input Variable

```
try_next_var
   if ( on last input variable ) then
      consid_inp_var := the first input variable
   else
      consid_inp_var := the next input variable
   end if
   phase := exploratory
   determine_exploratory_instructions
end try_next_var
```

A3.3 Linear Predictor Heuristic

A3.3.1 First Iteration Set-up

```
LP_DL_phase_setup
   phase := DL
   consid_inp_var := first input variable
   DL_base_point := closest point stored
   instructions := increase by 1
   generate_point := DL_base_point
end LP_DL_phase_setup
```

A3.3.2 Generator

```
LP_generator
   case ( modification direction of instructions ) is
   when increase =>
      execute_point := add instructions value to consid_inp_var of
         generate_point
   when decrease =>
      execute_point := subtract instructions value from
         consid_inp_var of generate_point
   end case
end LP_generator
```

A3.3.3 Evaluator

```
LP_evaluator
   if ( considered node traversed ) then
      heuristic is successful
   else
      case ( phase ) is
      when DL =>
         case ( traversal_effect of execute_point ) is
            when sibling_traversal =>
               DL_sibling_traversal_evaluator
            when upper_deviation =>
               DL_upper_deviation_evaluator
         end case
      when predictor =>
         case ( traversal_effect of execute_point and
               closeness to expected boundary of execute_point
               compared to closest point found ) is
         when sibling_traversal and closer =>
            predictor_closer_sibling_traversal_evaluator
         when sibling_traversal and further =>
            predictor_further_sibling_traversal_evaluator
         when upper_deviation =>
            predictor_upper_deviation_evaluator
         end case
      when creeper =>
         case ( traversal_effect of execute_point ) is
         when sibling_traversal =>
            creeper_sibling_traversal_evaluator
         when upper_deviation =>
            creeper_upper_deviation_evaluator
         end case
      end case
   end if
end LP_evaluator
```

A3.3.3.1 Determine-linearity Phase

```
DL_sibling_traversal_evaluator
   if ( just executed increase point ) then
      DL_increase_point := execute_point
      instructions := decrease by 1
      generate_point := DL_base_point
   elsif ( just executed decrease point ) then
      DL_decrease_point := execute_point
      if ( consid_inp_var influential on considered predicate )
            then
         attempt_to_predict_a_point
      else
         try_next_var
      end if
   end if
end DL_sibling_traversal_evaluator

attempt_to_predict_a_point
   if ( variations in input var value and predicate value linear )
         then
      predict_a_point_to_cause_just_sibling_traversal
   else
      creeper_setup
   end if
end attempt_to_predict_a_point

predict_a_point_to_cause_just_sibling_traversal
   {  Predict a point that will be close to the expected boundary
      and will cause sibling_traversal }
   { Firstly predict ( extrapolate ) value }
   phase := predictor
   predicted_input_value := extrapolation using DL_base_point
         and DL_increase_point
   generate_point := execute_point with predicted_input_value for
         consid_inp_var
   { Then modify value for just sibling_traversal }
   case ( predicted_input_value compared to consid_inp_var
         value of DL_base_point ) is
   when predicted less than base =>
      instructions := increase by 1
   when predicted greater than base =>
      instructions := decrease by 1
   when predicted equal to base =>
      try_next_var
   end case
end predict_a_point_to_cause_just_sibling_traversal

DL_upper_deviation_evaluator
   try_next_var
end DL_upper_deviation_evaluator

creeper_setup
   {  Creep to the expected boundary }
   phase := creeper
   case ( DL direction to come closer to the expected boundary )
      is
   when increase =>
      instructions := increase by 1
      generate_point := DL_increase_point
   when decrease =>
      instructions := decrease by 1
      generate_point := DL_decrease_point
   when neither =>
      try_next_var
   end case
end creeper_setup
```

A3.3.3.2 Predictor Phase

```
predictor_closer_sibling_traversal_evaluator
   if ( there has been an upper_deviation from the execution of
         the predicted point onwards ) then
      try_next_var
   else
      if ( the predicted point has just been executed ) then
         reverse the modification direction
      else
         { Continue with the present modification direction}
      end if
   end if
end predictor_closer_sibling_traversal_evaluator

predictor_further_sibling_traversal_evaluator
   try_next_var
end predictor_further_sibling_traversal_evaluator

predictor_upper_deviation_evaluator
   if ( there has been sibling_traversals from the execution of
         the predicted point onwards ) then
      try_next_var
   else
      if ( reached upper_deviation limit for one input variable )
            then
         try_next_var
      else
         { Continue modification in direction away from expected
            boundary }
      end if
   end if
end predictor_upper_deviation_evaluator
```

A3.3.3.3 Creeper Phase

```
creeper_sibling_traversal_evaluator
   if ( execute_point closer to expected boundary than closest
         point found) then
      { Continue with present modification direction }
   else
      try_next_var
   end if
end creeper_sibling_traversal_evaluator

creeper_upper_deviation_evaluator
   try_next_var
end creeper_upper_deviation_evaluator
```

A3.3.3.4 Abandoning Consideration of an Input Variable

```
try_next_var
   if ( heuristic failure criteria met ) then
      terminate operation of heuristic
   else
      if ( on last input variable ) then
         consid_inp_var := first input variable
      else
         consid_inp_var := next input variable
      end if
      DL_base_point := closest point found
      instructions := increase by 1
      phase := DL
      generate_point := DL_base_point
   end if
end try_next_var
```

A3.4 Boundary Follower Heuristic

A3.4.1 First Iteration Set-up

```
BF_OCP_phase_setup
   consid_inp_var := first input variable
   OCP_DL_base_point := closest point found
   instructions := increase by 1
   phase := OCP.DL
   generate_point := OCP_DL_base_point
end BF_OCP_phase_setup
```

A3.4.2 Generator

```
BF_generator
   case ( instructions ) is
   when increase =>
      execute_point := add instructions value to consid_inp_var of
            generate_point
   when decrease =>
      execute_point := subtract instructions value from
            consid_inp_var of generate_point
   when opposite direction to follow and increase cross =>
      case ( follow direction ) is
      when decrease =>
         FB_follow_var of execute_point := add instructions value
               to FB_follow_var of generate_point
      when increase =>
         FB_follow_var of execute_point := subtract instructions
               value from FB_follow_var of generate_point
      end case
      FB_cross_var of execute_point := add instructions value to
            FB_cross_var of generate_point
   end case
end BF_generation
```

A3.4.3 Evaluator

```
BF_evaluator
   if ( considered node traversed ) then
      heuristic is successful
   else
      case ( phase ) is
      when OCP =>
         if ( execute_point crossed boundary ) then
            OCP_try_next_var
         else
            case ( OCP subphase ) is
            when DL =>
               OCP_DL_evaluator
            when predictor =>
               OCP_predictor_evaluator
            when creeper =>
               OCP_creeper_evaluator
            end case
         end if
      when DIFCD =>
         DIFCD_evaluator
      when FB =>
         FB_evaluator
      when RBF =>
         RBF_evaluator
      end case
   end if
end BF_evaluator
```

A3.4.3.1 Obtain-a-close-point Phase

```
OCP_DL_evaluator
   if ( just executed increase point ) then
      OCP_DL_increase_point := execute_point
      instructions := decrease by 1
      generate_point := DL_base_point
   elsif ( just executed decrease point ) then
      OCP_DL_decrease_point := execute_point
      attempt_to_predict_a_point
   end if
end OCP_DL_evaluator
```

The procedure attempt_to_predict_a_point and the two procedures it calls, predict_a_point_to_cause_just_sibling_traversal and creeper_setup, are shown in appendix A3.3.3.1. In the LP's code however, a phase change should be read as a BF Obtain-a-close-point subphase change and a call to the procedure try_next_var, refers to the procedure OCP_try_next_var.

```
OCP_predictor_evaluator
   { Creep to and over the boundary located }
   case ( predicted value compared to consid_inp_var value of
         DL_base_point ) is
   when predicted less than base =>
      instructions := decrease by 1
   when predicted greater than base =>
      instructions := increase by 1
   end case
end OCP_predictor_evaluator
```

```
OCP_creeper_evaluator
   if ( execute_point closer to expected boundary than closest
         point stored ) then
      { Continue with present modification direction }
   else
      OCP_try_next_var
   end if
end OCP_creeper_evaluator

OCP_try_next_var
   if ( all input variables considered ) then
      { Setup for DIFCD phase }
      phase := DIFCD
      DIFCD_central_point := closest point stored
      generate_point := DIFCD_central_point
   else
      { Setup for DL subphase on next var }
      phase := OCP.DL
      consid_inp_var := next input variable
      DL_base_point := closest point stored
      generate_point := DL_base_point
      instructions := increase by 1
   end if
end OCP_try_next_var
```

A3.4.3.2 Determine-initial-follow-and-cross-details Phase

```
DIFCD_evaluator
   if ( done all DIFCD test procedure executions) then
      DIFCD_evaluate_and_allocate_roles_for_FB_phase
   else
      { Continue DIFCD test procedure executions }
      if ( completed test procedure executions on one input
            variable ) then
         consid_inp_var := next input variable
         instructions := increase by 1
      else
         instructions := decrease by 1
      end if
      generate_point := DIFCD_central_point
   end if
end DIFCD_evaluator

DIFCD_evaluate_and_allocate_roles_for_FB_phase
   DIFCD_evaluate_and_allocate_roles
   if ( both roles have been allocated ) then
      { Setup for FB phase }
      phase := FB
      FB_follow_var := DIFCD_follow_allocation
      FB_cross_var := DIFCD_cross_allocation
      FB_move_kind := follow
      instructions := increase by 1
      generate_point := DIFCD_central_point
      consid_inp_var := FB_follow_var
   else
      terminate operation of heuristic
   end if
end DIFCD_evaluate_and_allocate_roles_for_FB_phase
```

```
DIFCD_evaluate_and_allocate_roles
   for var := first input var to last input var loop
      DIFCD_evaluate_and_allocate_roles_for_each_var
   end for
   { Try to allocate follow role if not allocated }
   if ( follow role not allocated ) then
      if ( reserve follow allocated ) then
         DIFCD_follow_allocation :=
            DIFCD_reserve_follow_allocation
      end if
   end if
end DIFCD_evaluate_and_allocate_roles

DIFCD_evaluate_and_allocate_roles_for_each_var
   { Evaluate each var traversal_effects and try to allocate roles
         }
   if ( increase and decrease point for var caused no change in
         the considered predicate value ) then
      { no allocation }
   elsif ( increase and decrease point for var caused upper_
         deviation ) then
      { no allocation }
   else
      case ( DIFCD_central_point predicate value compared to
            expected boundary predicate value ) is
      when greater than =>
         case ( var's traversal_effect suitability) is
         when cross role =>
            try_to_allocate_cross_role
         when follow role =>
            try_to_allocate_follow_role
         end case
      when less than =>
         case ( var's traversal_effect suitability) is
         when cross role =>
            try_to_allocate_cross_role
         when follow role =>
            try_to_allocate_follow_role
         end case
      end case
   end if
end DIFCD_evaluate_and_allocate_roles_for_each_var

try_to_allocate_cross_role
   if ( cross not allocated ) then
      DIFCD_cross_allocation := var
   elsif ( reserve follow not allocated ) then
      DIFCD_reserve_follow_allocation := var
   end if
end try_to_allocate_cross_role

try_to_allocate_follow_role
   if ( follow not allocated ) then
      DIFCD_follow_allocation := var
   end if
end try_to_allocate_follow_role
```

A3.4.3.3 Follow-boundary Phase

```
FB_evaluator
   case ( FB_move_kind ) is
   when follow =>
      FB_follow_evaluator
   when cross =>
      FB_cross_evaluator
   end case
end FB_evaluator
```

```
FB_follow_evaluator
   { Change over to cross from follow move end point }
   FB_move_kind := cross
   consid_inp_var := FB_cross_var
   define_cross_direction
   generate_point := execute_point
end FB_follow_evaluator

define_cross_direction
   if ( a cross move has crossed the boundary during the present
         FB phase ) then
      if ( last follow move crossed the boundary ) then
         { Direction remains the same as the last cross move }
      else
         { Direction is the opposite of the last cross move }
         case ( last cross move direction ) is
         when increase =>
            instructions := decrease by 1
         when decrease =>
            instructions := increase by 1
         end case
      end if
   else
      { Boundary not crossed yet - direction is increase }
      instructions := increase by 1
   end if
end define_cross_direction

FB_cross_evaluator
   if ( execute_point crossed the boundary ) then
      { Changeover to follow from execute_point }
      FB_move_kind := follow
      consid_inp_var := FB_follow_var
      instructions := follow direction by 1
   elsif ( maximum cross step length reached ) then
      { Setup for RBF }
      phase := RBF
      instructions := opposite direction to follow and increase
            cross by 1
      generate_point := last follow move end point
   else
      if ( a cross move has crossed the boundary during the
            present FB phase ) then
         { Continue with a unidirectional cross search }
         instructions := same direction as last cross with
               increased step size
         generate_point := last follow move end point
      else
         { Continue with a bidirectional cross search }
         case ( cross direction ) is
         when increase =>
            instructions := decrease by same step size
         when decrease =>
            instructions := increase with increased step size
         end case
         generate_point := last follow move end point
      end if
   end if
end FB_cross_evaluator
```

A3.4.3.4 Reorient-boundary-follower Phase

```
RBF_evaluator
   if ( a search pair has been executed ) then
      search_pair_analysis
   else
      { setup for the decrease point from the central line }
      instructions := decrease by 1
      generate_point := cross variable value on central line with
            remaining input variables held constant
   end if
end RBF_evaluator
```

```
RBF_evaluator
   if ( a search pair has been executed ) then
      search_pair_analysis
   else
      { setup for the decrease point from the central line }
      instructions := decrease by 1
      generate_point := cross variable value on central line with
            remaining input variables held constant
   end if
end RBF_evaluator
```

```
setup_for_FB_phase
   phase := FB
   FB_move_kind := follow
   temp_var := FB_follow_var
   FB_follow_var := FB_cross_var
   FB_cross_var := temp_var
   if ( both search points on other side of the boundary ) then
      { follow direction remains unchanged }
   else
      follow direction is the same as the direction used to
            generate the point that crossed the boundary
   end if
   instructions := follow direction by 1
   generate_point := last follow move end point
end setup_for_FB_phase
```

References

Abbott, J. (1986) *Software Testing Techniques*, NCC Publications.

Adby, P.R. and Dempster, M.A.H. (1974) *Introduction to Optimisation Methods*, Chapman and Hall.

Andrews, D.M. and Benson, J.P. (1981) 'An Automated Program Testing Methodology and its Implementation'. In *5th Int. Conf. on Software Engineering*, IEEE, 254-261.

Barnes, J.G.P. (1989) *Programming in Ada*, Addison Wesley.

Basili, V.R. and Selby, R.W. (1987) 'Comparing the Effectiveness of Software Testing Strategies', *IEEE Transactions on Software Engineering*, **SE-13** (12), 1278-1296.

Beizer, B. (1983) *Software Testing Techniques*, Van Nostrand Reinhold.

Benson, J.P. (1981) 'Adaptive Search Techniques Applied to Software Testing', *ACM Performance Evaluation Revue*, **10** (1), 109-116.

Booch, G. (1987) *Software Components with Ada : Structures, Tools and Subsystems*, Benjamin/Cummings Pub. Co. Inc.

Box, M.J. (1965) 'A New Method of Constrained Optimization and a Comparison With Other Methods', *Computer Journal*, **8**, 42-52.

Box, M.J., Davies, D. and Swann, W.H. (1969) *Non-Linear Optimisation Techniques*, Oliver & Boyd.

Budd, T.A. (1981) 'Mutation Analysis : Ideas, Examples, Problems and Prospects'. In Chandrasekaran, B. and Radicchi, S. (eds.), *Computer Program Testing*, North Holland Publishing Co., 129-148.

Burgess, C.J. (1993) 'Software Testing Using an Automatic Generator of Test Data'. In Ross, M., Brebbia, C. A., Staples, G. and Stapleton, J. (eds.),*First Int. Conf. on Software Quality Management*, Southampton, UK, March 30 - April 1, 541-556.

Cimittle, A. and Carlini, U.D. (1991) 'Reverse Engineering : Algorithms for Program Graph Production', *Software Practice and Experience*, **21** (5), 519-537.

Clarke, L.A. (1976) 'A System to Generate Test Data and Symbolically Execute Programs', *IEEE Transactions on Software Engineering*, **SE-2** (3), 215-222.

Cohen, E.I. (1978) *A Finite Domain-Testing Strategy for Computer Program Testing*, PhD, Ohio State University, Columbus.

Conn, A.R., Gould, N.I.M. and Toint, P.L. (1994) *Large-Scale Nonlinear Constrained Optimization : A Current Survey*, CERFACS, Toulouse Cedex, France.

Cooper, D.W. (1976) 'Adaptive Testing'. In *2nd Int. Conf. on Software Engineering*, San Francisco, US, October 13-15, 102-105.

Coward, P.D. (1988) 'Symbolic Execution Systems - A Review', *Software Engineering Journal*, November, 229-239.

Cross, J.H., Chang, K.H., Carlisle, W.H. and Brown, D.B. (1991) 'Expert System Assisted Test Data Generation For Software Branch Coverage', *Data and Knowledge Engineering*, **6** , 279-295.

Deason, W.H., Brown, D.B., Kai-Hsiung, C. and II, J.H.C. (1991) 'A Rule-Based Software Test Data Generator', *IEEE Transactions on Knowledge and Data Engineering*, **3** (1), 108-117.

DeMillo, R.A. and Offutt, A.J. (1991) 'Constraint-Based Automatic Test Data Generation', *IEEE Transactions on Software Engineering*, **17** (9), 900-910.

DeMillo, R.A., Lipton, R.J. and Sayward, F.G. (1978) 'Hints on Test Data Selection : Help for the Practicing Programmer', *IEEE Computer*, **11** (4), 43-41.

Denney, R. (1991) 'Test-Case Generation From Prolog-Based Specifications', *IEEE Software*, **8** (2), 49-57.

Deutsch, M.S. (1982) *Software Verification and Validation : Realistic Project Approaches*, Prentice-Hall.

Duran, J.W. and Ntafos, S.C. (1984) 'An Evaluation of Random Testing', *IEEE Transactions on Software Engineering*, **SE-10** (4), 438-444.

Elshoff, J.L. (1976) 'An Analysis of Some Commercial PL/1 Programs', *IEEE Transaction on Software Engineering*, **SE-2** (2), 113-120.

Fagan, M.E. (1976) 'Design and Code Inspections to Reduce Errors in Program Development', *IBM Systems Journal*, **15** (3), 182-211.

Furukawa, Z. and Ushijima, K. (1987) 'A Model of the Testing Support Method With Sequences of a Directed Graph'. In *Eleventh Ann. Int. Computer Software and Applications Conference (COMPSAC 87)*, Tokyo, Japan, October 7-9, IEEE, 311-316.

Furukawa, Z., Nogi, K. and Tokunaga, K. (1985) 'AGENT : An Advanced Test-Case Generation System for Functional Testing'. In Wojcik, A. S. (ed.), *National Computer Conference*, Chicago, Illinois, July 15-18, AFIPS Press, 525-535.

Gallagher, M.J. and Narasimhan, V.L. (1993) 'A Software System for the Generation of Test Data for Ada Programs', *Microprocessing and Microprogramming*, **38** , 637-644.

Gill, P.E. and Murray, W. (1979) 'Performance Evaluation for Optimisation Software'. In Fosdick, L. (ed.), *Performance Evaluation of Numerical Software*, IFIP, North-Holland Pub. Co., 221-234.

Gill, P.E. and Murray, W. (eds.) (1974) *Numerical Methods for Constrained Optimisation*, Academic Press Inc.

Glass, H. and Cooper, L. (1965) 'Sequential Search : A Method for Solving Constrained Optimisation Problems', *Journal for the Association for Computing Machinery*, **12** (1), 71-82.

Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison Wesley.

Graham, D.R. (1991) *Computer Aided Software Testing : CAST Report*, Unicom.

Hecht, M.S. (1977) *Flow Analysis of Computer Programs*, Elsevier North-Holland Inc.

Hedley, D. and Hennell, M.A. (1985) 'The Causes and Effects of Infeasible Paths in Computer Programs'. In *Eighth Int. Conf. on Software Engineering*, London, August 28-30, IEEE, 259-266.

Holmes, S.T., Jones, B.F. and Eyres, D.E. (1993) 'An Improved Strategy for the Automatic Generation of Test Data'. In Ross, M., Brebbia, C. A., Staples, G. and Stapleton, J. (eds.), *First Int. Conf. on Software Quality Management*, Southampton, UK, March 30 - April 1, 565-577.

Hooke, R. and Jeeves, T.A. (1961) '"Direct Search" Solution of Numerical and Statistical Problems', *Journal for the Association for Computing Machinery*, **8** , 212-229.

Howden, W.E. (1977) 'Symbolic Testing and the DISSECT Symbolic Evaluation System', *IEEE Transactions on Software Engineering*, **SE-3** (4), 266-278.

Howden, W.E. (1987) *Functional Program Testing and Analysis*, McGraw-Hill.

Huang, J.C. (1978) 'Program Instrumentation and Software Testing', *IEEE Computer, April*, **11** , 25-32.

Inamura, H. (1989) 'Trial-and-Error Method for Automated Test Data Generation and Its Evaluation', *Systems and Computers in Japan*, **20** (2), 78-91.

Ince, D.C. (1987) 'The Automatic Generation of Test Data', *The Computer Journal*, **30** (1), 63-69.

Infotech (1979) *Infotech State of the Art Report : Vol.1 Analysis and Bibliography*, Infotech Int. Ltd.

Klingman, W.R. and Himmelblau, D.M. (1964) 'Nonlinear Programming With the Aid of Multiple Gradient Summation Techniques', *Journal of the Association for Computing Machinery*, **11**, 400-415.

Knuth, D.E. (1971) 'An Empirical Study of FORTRAN Programs', *Software-Practice and Experience*, **1**, 105-133.

Korel, B. (1990a) 'Automated Software Test Data Generation', *IEEE Transactions on Software Engineering*, **16** (8), 870-879.

Korel, B. (1990b) 'A Dynamic Approach of Test Data Generation'. In *Conf. on Software Maintenance*, San Diego, US, November 26-29, IEEE, 311-316.

Korel, B. (1992) 'Dynamic Method for Software Test Data Generation', *Journal of Software Testing, Verification and Reliability*, **2**, 203-213.

Kundu, S. (1979) 'SETAR - A New Approach to Test Case Generation'. In Westley, A. (ed.), *Infotech State of the Art Report Software Testing : Vol. 2. Invited Papers*, Infotech Int. Ltd., 161-186.

McMullin, P.R. and Gannon, J.D. (1983) 'Combining Testing With Formal Specifications : A Case Study', *IEEE Transactions on Software Engineering*, **SE-9** (3), 328-335.

Miller, W. and Spooner, D.L. (1976) 'Automatic Generation of Floating-Point Test Data', *IEEE Transactions on Software Engineering*, **SE-2** (3), 223-226.

Minoux, M. (1986) *Mathematical Programming : Theory and Algorithms*, Wiley.

Moranda, P.B. (1978) 'Limits to Program Testing With Random Number Inputs'. In *COMPSAC '78*, 521-526.

Murray, W. (ed.) (1972) *Numerical Methods for Unconstrained Optimisation*, Academic Press.

Myers, G.J. (1979) *The Art of Software Testing*, Wiley-Interscience.

Nelder, J.A. and Mead, R. (1965) 'A Simplex Method for Function Minimization', *The Computer Journal*, **7**, 308-313.

Neumann, P.G. (1995) 'Inside RISKS / Risks to the Public from Computers and Related Systems', *ACM SIGSOFT Software Engineering Notes*, **20**.

Ntafos, S.C. (1988) 'A Comparison of Some Structural Testing Strategies', *IEEE Transactions on Software Engineering*, **14** (No 6), 868-874.

Offutt, A.J. (1992) 'Investigations of the Software Testing Coupling Effect', *ACM Transactions on Software Engineering and Methodology*, **1** (1), 5-20.

Offutt, A.J. and Seaman, E.J. (1990) 'Using Symbolic Execution to Aid Automatic Test Data Generation'. In *COMPASS '90*, 12-21.

Prather, R.E. and Myers, J.P. Jr. (1987) 'The Path-Prefix Software Testing Strategy', *IEEE Transactions on Software Engineering*, **SE-13** (7), 761-766.

Pressman, R.S. (1994) *Software Engineering : A Practitioner's Guide, 3rd Ed.*, McGraw Hill.

Rabinowitz, F.M. (1995) 'Algorithm 744: A Stochastic Algorithm for Global Optimization with Constraints', *ACM Transactions on Mathematical Software*, **21** (2), 194-213.

Ramamoorthy, C.V., Ho, S.F. and Chen, W.T. (1976) 'On the Automated Generation of Program Test Data', *IEEE Transactions on Software Engineering*, **SE-2** (4), 293-300.

Richardson, D.J. and Clarke, L.A. (1981) 'A Partition Analysis Method to Increase Program Reliability'. In *Fifth Int. Conf. on Software Engineering*, IEEE, 244-253.

Roberts, S.M. and Lyvers, H.I. (1961) 'The Gradient Method in Process Control', *Industrial and Engineering Chemistry*, **53** (11), 877-882.

Roper, M. (1994) *Software Testing*, McGraw-Hill.

Roper, M., Maclean, I., Brooks, A., Miller, J. and Wood, M. (1995) *Genetic Algorithms and the Automatic Generation of Test Data*, Dept. Computer Science, University of Strathclyde, Glasgow, UK.

Rosenbroch, H.H. (1960) 'An Automatic Method for Finding the Greatest or Least Value of a Function', *The Computer Journal*, **3**, 175-184.

Schmitz, P., Megan, R.v. and Bons, H. (1980) 'Methods of Systematic Test Case Determination and Test Case Preparation'. In Ebert, R., Lugger, J. and Goecke, L. (eds.), *Practice in Software Adaption and Maintenance*, North-Holland Pub. Co., 209-221.

Tai, K.C. (1990) 'Condition-Based Software Testing Strategies'. In *Fourteenth Ann. Int. Computer Software and Applications Conference (COMPSAC 90)*, Vol. Chicago, Illonois, US, October 29 - November 2, IEEE, 564-569.

Voges, U., Gmeiner, L. and Mayrhauser, A.Av. (1980) 'SADAT - An Automated Testing Tool', *IEEE Transactions on Software Engineering*, **SE-6** (3), 286-290.

Weyuker, E.J. (1982) 'On Testing Non-Testable Programs', *The Computer Journal*, **25** (4), 465-470.

Weyuker, E.J. (1986) 'Axiomatizing Software Test Data Adequacy', *IEEE Transactions on Software Engineering*, **SE-12** (12), 1128-1138.

White, L.J. and Cohen, E.I. (1979) 'A Domain Strategy for Computer Program Testing'. In Westley, A. (ed.), *Infotech State of the Art Report : Vol 2. Invited Papers*, Infotech Int. Ltd., 325-363.

White, L.J. and Cohen, E.I. (1980) 'A Domain Strategy for Computer Program Testing', *IEEE Transactions on Software Engineering*, **SE-6** (3), 247-257.

Woodward, M.R., Hedley, D. and Hennell, M.A. (1980) 'Experience With Path Analysis and Testing of Programs', *IEEE Transactions on Software Engineering*, **SE-6** (3), 278-285.

Yau, S.S. and Grabow, P.C. (1981) 'A Model for Representing Programs Using Hierarchical Graphs', *IEEE Transactions on Software Engineering*, **SE-7** (6), 556-574.