Evolutionary Search for Test Data in the Presence of State Behaviour

by

Philip McMinn

Submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy at
the University of Sheffield
Department of Computer Science

January 2005

Abstract

The application of metaheuristic search techniques, such as evolutionary algorithms, to the problem of automatically generating software test data has been a burgeoning interest for many researchers in recent years. To date, work in applying search techniques to structural test data generation has largely focused on generating inputs for test objects with input-output behaviour. This thesis aims to extend the approach for test objects with state behaviour. This presents several challenges, not least because test goals with state-based test objects may require input sequences to be generated. Another problem includes generating test data in the presence of internal variables such as flags, enumerations and counters. Such variables are often responsible for managing the "state" of the test object. However, their use can lead to information loss with regards to the original input conditions that lead to the fulfilment of certain test goals. Consequently the search receives less guidance, and may fail to find test data.

This thesis proposes an extended evolutionary structural test data generation approach that allows input sequences to be generated, and tackles internal variable problems through hybridization of the method with an extended *chaining approach*. The basic idea of the chaining approach is to find a sequence of statements, involving internal variables, which need to be executed prior to the test goal. By requiring that these statements are executed, information previously unavailable to the search can be made use of, possibly guiding it into potentially promising and unexplored areas of the test object's input domain.

A number of experiments show the value of the approach for both test objects with states and test objects with input-output behaviour. For all test objects considered, higher levels of branch coverage are obtained.

Acknowledgements

I would like to thank my supervisor Professor Mike Holcombe for providing this opportunity, and for his support and advice throughout this PhD. I would also like to thank DaimlerChrysler Research and Technology for funding this work, and in particular Dr Joachim Wegener, André Baresel and Dr Harmen Sthamer for their helpful support and comments throughout this project. Thanks also go to Professors Mark Harman and Rob Hierons for setting up and organizing the EPSRC-funded SEMINAL and FORTEST networks respectively, which have provided an invaluable base for meeting other researchers in this field, and for presenting and receiving feedback on this work. Thanks in particular to Professor Mark Harman, Professor John Clark and Dr Bogdan Korel for useful and helpful discussions. I would also like to thank the various anonymous referees for their comments on papers submitted for publication as a result of this work, as their comments and feedback have been extremely beneficial.

I would like to thank my wife, Rebecca McMinn, for her love, support and understanding during the three years of this project. Also to my friends for life outside research, including Andy Todd, Phil Harper, John Hobson, Rob Monro, Sam Chapman, Tony Hart and others, and members of Christ Church Central Sheffield. Thanks to my parents, family, and in-laws for their ongoing love and support. Finally thanks to God, my Lord and Saviour; for everything I am able to achieve, and for everything that alone, I cannot achieve.

Declaration

The work presented in this thesis is original work undertaken between October 2001 and October 2004 at the University of Sheffield. Some of this work has been published elsewhere:

- P. McMinn, M. Holcombe, The State Problem for Evolutionary Testing. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Chicago, USA. Lecture Notes in Computer Science, Volume 2724, pp. 2488-2500, Springer Verlag, 2003.

- P. McMinn, Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2), pp. 105-156, 2004.

- P. McMinn, M. Holcombe, Hybridizing Evolutionary Testing with the Chaining Approach. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Seattle, USA. Lecture Notes in Computer Science, Volume 3103 pp. 1363-1374, Springer Verlag, 2004. Winner of Best Paper Award for Search-Based Software Engineering Track.

Contents

1 Introduction 1

1.1 Metaheuristic Search Techniques 1

1.2 Applying Search Techniques to Test Data Generation 4

1.2.1 The Problem of Internal Variables 6

1.3 The Problem of this Thesis: States and Evolutionary Test Data Generation 8

1.4 Aims and Objectives of this Thesis 8

1.5 Contributions of this Thesis 9

1.6 Overview of the Structure of this Thesis 10

2 Search-Based Software Test Data Generation: A Literature Review 13

2.1 Introduction 13

2.2 Metaheuristic Search Techniques 13

2.2.1 Hill Climbing 14

2.2.2 Simulated Annealing 15

2.2.3 Evolutionary Algorithms 16

2.3 Structural (White-Box) Testing 20

2.3.1 Basic Concepts 21

2.3.2 Static Structural Test Data Generation 22

2.3.3 Dynamic Structural Test Data Generation 28

2.3.4 Applying Simulated Annealing 38

2.3.5 Applying Evolutionary Algorithms 40

2.4 Functional (Black-Box) Testing 55

2.4.1 Generating Test Data from a Z Specification 55

2.4.2 Testing Specification Conformance 56

2.5 Grey-Box Testing 60

2.5.1 Assertion Testing 61

2.5.2 Exception Condition Testing 62

2.6 Non-Functional Testing 63
2.6.1 Execution Time Testing 63
2.7 Conclusions 66

3 The State Problem 69
3.1 Introduction 69
3.2 The Challenges Caused by States in Test Objects 69
3.2.1 Input Sequences 70
3.2.2 The Internal Variable Problem 70
3.3 Experimental Study 1 - State-Based Experiments with the Standard Evolutionary Approach 75
3.3.1 Test Objects 76
3.3.2 Experimental Setup 80
3.3.3 Results 82
3.3.4 Conclusions of Experimental Study 1 87
3.4 Relevant Literature for Evolutionary Input Sequence Generation 88
3.4.1 Generation of Input Sequences for Single Function Test Objects with State Behaviour 88
3.4.2 Evolutionary Structural Testing of Classes 89
3.5 An Approach for the Generation Input Sequences for Multiple Function Test Objects 91
3.5.1 Modified Encoding 91
3.6 Experimental Study 2 - State-Based Experiments with an Evolutionary Structure-Oriented Sequence Generation Approach . . 92
3.6.1 Experimental Setup 92
3.6.2 Results 93
3.6.3 Conclusions of Experimental Study 2 100
3.7 Conclusions 101

4 Revisiting the Problem of Internal Variables 103
4.1 Introduction 103
4.2 Solutions to the Flag Problem 104
4.3 Use of a State-based Model of the System 104
4.3.1 Reverse Engineering State Models for Testing 104
4.3.2 Reverse Engineering State Models for Model Checking . . 107
4.4 The Chaining Approach 108
4.4.1 The Concept of a "Last Definition" 110
4.4.2 Event Sequences 112
4.4.3 The Chaining Process 112

4.4.4 Formal Generation of an Event Sequence 114
4.4.5 Repetition of a Problem Node 119
4.4.6 Test Data Search 119
4.4.7 Test Data Generation for Covering Branches and Definition-Use Pairs 122
4.4.8 Evaluation of the Chaining Approach 123
4.5 Conclusions 123

5 Hybridizing Evolutionary Test Data Generation with an Extended Chaining Approach 125
5.1 Introduction 125
5.2 The Hybrid Approach 125
5.2.1 Test Data Search 126
5.2.2 Identification of Problem Nodes and Variables Used at the Problem Node 127
5.2.3 Extensions to the Original Chaining Approach 129
5.3 Experimental Study 3 - Test Data Search using the Hybrid Approach 135
5.3.1 Test Objects 136
5.3.2 Experimental Setup 144
5.3.3 Results 146
5.3.4 Conclusions of Experimental Study 3 164
5.4 Conclusions 164

6 Extension of the Hybrid Approach for the State Problem 167
6.1 Introduction 167
6.2 Extending the Hybrid Approach 167
6.2.1 Finding Last Definitions and Generating Event Sequences 167
6.2.2 Test Data Search 168
6.2.3 Chaining Tree Search 172
6.3 Experimental Study 4 - Test Data Generation using the Sequence Hybrid Approach 173
6.3.1 Experimental Setup 173
6.3.2 Results 174
6.4 Experimental Study 5 - An Industrial Test Object 180
6.4.1 Results 181
6.5 Conclusions 182

7 Conclusions and Future Work 183
7.1 Summary of Achievements . . . 183
7.1.1 Identification of Problems for the Standard Evolutionary Approach and State-Based Test Objects . . . 183
7.1.2 Solution Development for Improving Evolutionary Test Data Generation for State-Based Test Objects . . . 184
7.1.3 Stage One - Sequence Evolutionary Approach . . . 185
7.1.4 Stage Two - The Hybrid Approach . . . 185
7.1.5 Final Stage - The Sequence Hybrid Approach . . . 186
7.1.6 Overall Conclusions . . . 187
7.2 Restrictions and Limitations . . . 187
7.2.1 Types of Programs . . . 187
7.2.2 Scalability . . . 188
7.3 Summary of Future Work . . . 188
7.3.1 Improvement of Test Data Search in the Presence of Nesting and Short-Circuiting Logical Operators . . . 188
7.3.2 Static Analysis to Reduce the Size of the Chaining Tree . 190
7.3.3 Search Space Reduction using Variable Dependence Analysis . . . 190
7.3.4 Analysis of Assignments as a Result of Boolean Conditions 191
7.3.5 Possible Improvement of Efficiency via Seeding . . . 192
7.3.6 Possible Improvement of Test Data Search using Multi-Objective Optimization . . . 193
7.3.7 Heuristic Chaining Tree Search . . . 194

Appendices 195

A Experimental Framework 195
A.1 Introduction . . . 195
A.1.1 Overview of the System . . . 195
A.2 Preparation of Test Objects . . . 196
A.2.1 Inlining of Embedded Function Calls . . . 197
A.2.2 Parsing of Source Code to XML . . . 197
A.2.3 Creation of Internal Model of the Test Object Source Code 197
A.2.4 Test Object Instrumentation . . . 198
A.2.5 Addition of Reset Functionality . . . 198
A.2.6 Resetting Multiple Function Test Objects . . . 200
A.2.7 Compilation of Test Object to a Dynamic Link Library . 200
A.3 Implementation Details of the Test Data Search . . . 200

A.3.1 The Evolutionary Algorithm Toolbox 201
A.3.2 Decoding Individuals, Test Object Execution, and Condition Distance Feedback 201
A.3.3 Computing Branch Distance Values 202
A.4 Evolutionary Search Parameters 204

B Program Code for Synthetic Test Objects with State Behaviour 207
B.1 Anomaly Detector 207
B.2 Array Difference 210
B.3 Postcode 211
B.4 Sliding Window 215
B.5 Smoke Detector 219
B.6 Sortcode 221
B.7 Stack 224
B.8 Telephone Number 226
B.9 Vending Machine 228

Bibliography 233

List of Figures

1.1 Searching for $x^2 - 2x - 10 = 0$ 2
1.2 Hill climbing in an example search space 3
1.3 Simulated annealing has the potential to escape local optima . . 4
1.4 Evolutionary algorithms sample many points of the search space in a single step 5
1.5 An example C program and control flow graph for applying test data search 6
1.6 Branch distances as a result of a flag variable 7

2.1 High level description of a hill climbing algorithm 15
2.2 High level description of a simulated annealing algorithm 16
2.3 High level description of a genetic algorithm 21
2.4 A triangle classification program and its corresponding control flow graph 23
2.5 Control dependence graph for the triangle classification program 24
2.6 Example program with nested structures 32
2.7 Example program for demonstrating the goal-oriented approach . 34
2.8 Control flow graph and branch classification for example program demonstrating the goal-oriented approach 35
2.9 Example program for demonstrating the chaining approach . . . 36
2.10 Objective function landscape for execution of a node in the example program for demonstrating the chaining approach 36
2.11 Example program for comparing different objective functions . . 39
2.12 Classification of dynamic structural test data generation techniques using evolutionary algorithms 40
2.13 Example landscape using the objective function of Pargas *et al.* 43
2.14 Example landscape using the objective function of Tracey 43
2.15 Example landscape using the objective function of Wegener *et al.* 43
2.16 Program with a loop and its control flow graph 47
2.17 Program with unstructured control flow 48

2.18 Control graphs for program with unstructured control flow . . . 48
2.19 Example program with a flag variable 49
2.20 Objective function landscape for the example program with flag variable 49
2.21 Objective function landscape for the predicate `d == 0` for the example program with flag variable 50
2.22 Alternative version of the flag example program 50
2.23 Flag removed from branch predicates of alternative version of flag program via program transformation 51
2.24 Deceptive objective function example program 52
2.25 Landscape for the deceptive objective function example program 53
2.26 Wrapping counter example program 57
2.27 Objective function landscape for wrapping counter example . . . 58
2.28 Objective function landscapes for individual disjuncts of the wrapping counter example 59

3.1 Fragment of code from a stack module 71
3.2 The `check_evaluations` example, demonstrating information loss with regards to branch distances based on an internal variable . . 73
3.3 Information loss with regards to branch distances based on an internal counter state variable 74
3.4 Branch distance plot using the return value of the `full()` function against size of the stack, calculated using |1 − `full()`| . . . 75
3.5 Test sequence generation example 89
3.6 Generating a generic encoding and mapping the universal parameter set to individual function call signatures 92
3.7 Comparison of coverage levels for the standard evolutionary approach verses the sequence evolutionary approach 95
3.8 Comparison of success rate levels for the standard evolutionary approach verses the sequence evolutionary approach 95

4.1 The coinbox example 106
4.2 Object state diagram for the coinbox example 107
4.3 Example for demonstrating the chaining approach 109
4.4 Finding last definitions 111
4.5 Example for finding last definitions 111
4.6 Constraint set violation example 115
4.7 The chaining tree 115
4.8 The chaining process 116

4.9 Multiple problem node example 118
4.10 Problem node repetition example 120

5.1 Example program for calculating the objective value of an event sequence 128
5.2 Objective function landscape for example program 128
5.3 Example code for finding problem nodes 130
5.4 Recursive procedure for generating event sequences using influencing sets 133
5.5 Program code for the “Counter” test object 136
5.6 Program code for the “Deceptive” test object 137
5.7 Program code for the “Enumeration” test object 139
5.8 Program code for the “Flag Assignment” test object 140
5.9 Program code for the “Flag Avoid Assignment” test object . . . 140
5.10 Program code for the “Flag Avoid Loop Assignment” test object 141
5.11 Program code for the “Flag Loop Assignment” test object 141
5.12 Program code for the “Multiple Flag” test object 142
5.13 Program code for the “Multiple Flag 2” test object 143
5.14 Number of objective function evaluations for termination criteria “a” and “b” for successful searches 148
5.15 Number of objective function evaluations for termination criteria “a” and “b” for unsuccessful searches 152
5.16 Objective function landscapes and average best objective value plots for the “Counter” test object 154
5.17 Objective function landscapes and average best objective value plots for the “Deceptive” test object 155
5.18 Objective function landscapes and average best objective value plots for the “Enumeration” test object 157
5.19 Objective function landscapes and average best objective value plots for the “Flag Assignment” and “Flag Avoid Assignment” test objects 158
5.20 Objective function landscapes and average best objective value plots for the “Flag Avoid Loop Assignment” test object 160
5.21 Objective function landscapes and average best objective value plots for the “Flag Loop Assignment” test object 162
5.22 Objective function landscapes and average best objective value plots for the “Multiple Flag” test object 163
5.23 Objective function landscapes and average best objective value plots for the “Multiple Flag 2” test object 165

6.1 Example for finding last definitions for a one function test object with states 168
6.2 Example for finding last definitions for a multiple function test object with states 169
6.3 "Global" control flow graph for the test objects with state behaviour 169
6.4 Comparison of coverage levels for the sequence hybrid approach against the standard and sequence evolutionary approaches . . . 175
6.5 Comparison of success rate levels for the sequence hybrid approach against the standard and sequence evolutionary approaches 176

7.1 Development of the solution proposed by this thesis to the state problem 185
7.2 Example for future research with search space reduction 191
7.3 Example for future research with seeding 193
7.4 Example for future research with multi-objective optimization . . 194

A.1 Overview of ETState test data evaluation process 196
A.2 Example for tracing execution paths 202
A.3 GEATbx option file 206

List of Tables

2.1 Korel's objective functions for relational predicates 31
2.2 Tracey's objective functions for relational predicates 39
2.3 Tracey's objective functions for logical connectives 57

3.1 Technical details of the synthetic state-based test objects 76
3.2 Success rate and coverage for the experiments using the standard evolutionary approach and random test data generation 84
3.3 Evaluations for the experiments using the standard evolutionary approach and random test data generation 85
3.4 Search times for the experiments using the standard evolutionary approach and random test data generation 86
3.5 Function call sequence lengths used for each state-based test object used during the sequence generation experiments 93
3.6 Results for the experiments using the sequence evolutionary approach and random sequence approach 94
3.7 Evaluations for the sequence and random sequence generation experiments 97
3.8 Search times for the sequence and random sequence generation experiments 98
3.9 Earliest, latest and average generation of last individual offering improvement on the best objective value found in the sequence evolutionary approach test data searches for the branches of each test object 99

5.1 Success rate for each experimental setup 146
5.2 Average and maximum generations to a solution using termination criterion "a" 148
5.3 Number of objective function evaluations for each test object and experimental setup 149
5.4 Search times for each test object and experimental setup 150
5.5 Event sequences considered by each setup 151

6.1 Success rate and coverage using the sequence hybrid approach, compared to random test sequence generation 174
6.2 Average and maximum number of test sequence evaluations for searches using the sequence hybrid approach and random sequence generation 177
6.3 Average and maximum times for searches using the sequence hybrid approach and random sequence generation 178
6.4 Event sequences used for sequence hybrid approach searches . . . 178
6.5 Chaining tree depth reached by sequence hybrid approach searches 179
6.6 Success rate and coverage for the industrial experiment 181
6.7 Evaluations for the industrial experiment 181
6.8 Times for the industrial experiment 181
6.9 Event sequences attempted in the industrial experiment 182
6.10 Chaining tree depth reached for the industrial experiment 182

7.1 Description of evolutionary test data generation methods used in this thesis 184

Chapter 1

Introduction

Software testing is an important yet extremely laborious and costly process. It can account for up to 50% of software development budgets [Bei90], but yet adds nothing to the final product in terms of its functionality. In addition to the problem of cost, the very nature of testing being that of seeking faults [Mye79] makes it an essentially destructive process that is not particularly enjoyed by software engineers. If the testing process could be automated, labour effort could be reduced, and the costs of testing would fall.

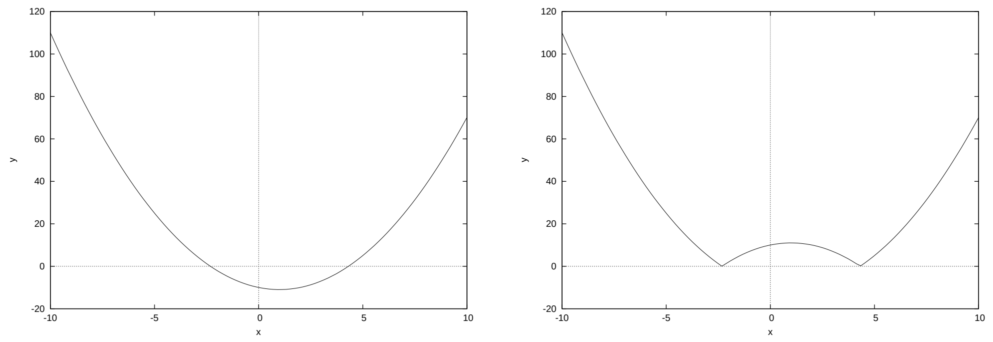
The subject of this thesis is the automatic generation of software test data. This is a difficult problem, because for a program of any realistic size, the input domain is too large to be searched exhaustively. On the other hand, random input generators are unlikely to thoroughly test unusual or exceptional features of a test object, which are only exercised by a small proportion of its overall input domain.

Pragmatic approaches, such as the derivation of test data through static analysis of the program's source code, are limited by the dynamic nature of software - for example the presence of unbounded loops and dynamic memory referencing, such as the use of pointers.

For these reasons, the application of *metaheuristic search techniques*, such as evolutionary algorithms, to search a program's input domain for test data has been a burgeoning interest for many researchers in recent years.

1.1 Metaheuristic Search Techniques

Metaheuristic search techniques are high-level frameworks which use heuristics to find solutions to problems without the need to perform a full exhaustive enumeration of a search space. In this way, solutions may be found to combinatorial problems at a reasonable computational cost. Such a problem may have been



(a) $x^2 - 2x - 10$ (b) cost function

Figure 1.1: Searching for $x^2 - 2x - 10 = 0$

classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical. These frameworks are not standalone algorithms in their own right, but rather "strategies" that are ready for adaptation to specific problems.

In order to adapt a metaheuristic search technique to a specific problem, a number of different decisions have to be made - for example the way in which solutions should be encoded so that they can be manipulated by the search. A good choice of encoding will ensure that similar solutions in unencoded space are also "neighbours" in representational space. In this way, the search will be allowed to move easily from one solution to another that has a similar set of properties in common. These movements are dependent on the evaluation of candidate solutions, performed using a problem-specific *objective function*. With feedback from the objective function, the search seeks "better" solutions based on knowledge and experience of previous ones. A good objective function is therefore critical to the success of the search. Solutions that are "better" in some respect should be rewarded with better objective values, whereas poorer solutions should be punished with poorer objective values. Whether a "better" objective value is, in practice, a higher value or lower value, is dependent on whether the search is seeking to minimize or maximize the objective function. An objective function which is being maximized reflects the relative "goodness" of candidate solutions, whereas an objective function to be minimized (sometimes referred to in this context as a *cost function*) reflects the relative undesirability of solutions.

Suppose, for example, the goal of the search is to find a root of the function $x^2 - 2x - 10$ (Figure 1.1a). The cost function might simply be $|x^2 - 2x - 10|$ (Figure 1.1b). A value of 7 has a better objective value than that of 10, since

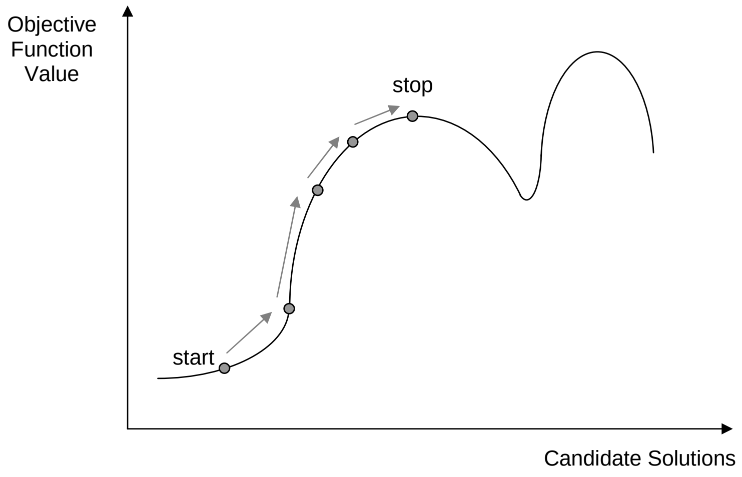


Figure 1.2: Hill climbing in an example search space

the output of the function is closer to approaching the positive root at 4.32. The search is encouraged to search around the value of 7, possibly encountering further "better" values, for example the values 5 or 6.

Hill climbing is arguably the simplest of metaheuristic search techniques. Starting with a random point in the search space, the current solution is improved by iteratively jumping to better ones found in its neighbourhood. This progressional improvement is likened to the climbing of hills in the "landscape" of a maximizing objective function. In this landscape, peaks characterize solutions with locally optimal objective values, and troughs signify solutions with the locally poorest objective values. A problem with local search is its tendency to get stuck in local optima - points in the search space that represent the best solutions in their neighbourhood, but are not in fact, the globally best solutions (Figure 1.2).

Simulated annealing attempts to alleviate this problem by moving around the search space more freely, with the potential to descend from hills in the objective function in order to explore new, potentially higher ones (Figure 1.3). This freedom is governed by a parameter known as the "temperature" of the search. As the temperature cools, the degree of freedom is reduced, until the search enters a final hill climbing phase.

Evolutionary algorithms are quite different to local searches such as hill climbing and simulated annealing, in that they maintain a *population* of solutions, rather than just one solution. During the search, the solutions are recombined with one another, with a hope of producing new offspring solutions

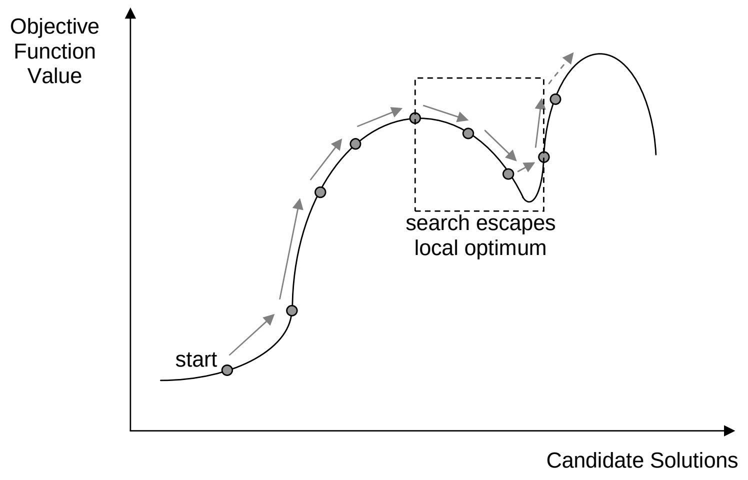


Figure 1.3: Simulated annealing has the potential to escape local optima

that have a blend of the best characteristics of their parents. Evolutionary algorithms sample several different points of the search space in a single step (Figure 1.4), and are therefore referred to as *global* search techniques. Global search techniques tend to be more robust than local search methods (such as hill climbing or simulated annealing) in objective function landscapes that contain local optima or plateaux.

Evolutionary algorithms are the search method of choice of this thesis. The application of evolutionary algorithms to test data generation is referred to throughout this thesis as *evolutionary test data generation*. In the literature, the use of evolutionary algorithms to automate the testing process is more generally referred to as *evolutionary testing*.

1.2 Applying Search Techniques to Test Data Generation

The application of search techniques to test data generation requires that inputs can be encoded in such a manner that they can be manipulated by the search technique. Furthermore, an objective function is required so that the search technique can find the appropriate test data. The "goodness" of candidate test data inputs is often expressed in terms of the "closeness" that the input data is to fulfilling the test goal. The measure of closeness differs depending on whether the test strategy is of a structural, functional or non-functional nature.

Structural or white-box testing strategies require the coverage of a certain type of structure in the code of a program, for example all of its statements or

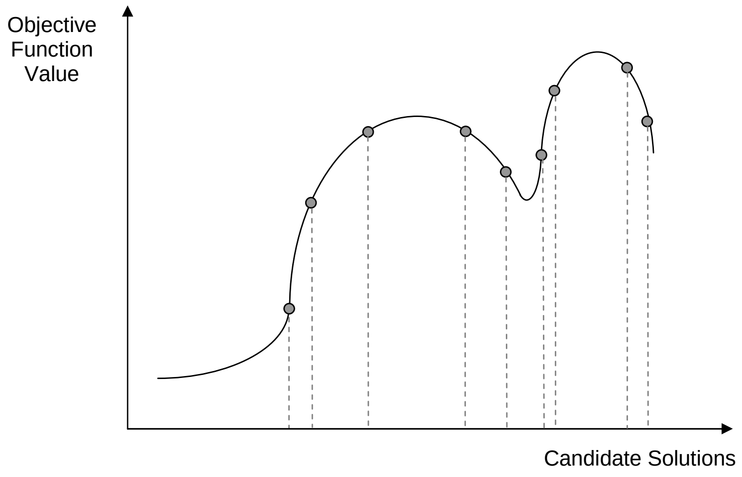


Figure 1.4: Evolutionary algorithms sample many points of the search space in a single step

all of its branches. The objective functions used by researchers for this purpose reward inputs that come close to executing each desired structure, and punish those that are far away. This closeness can be measured in different ways, for example on the basis of the control flow graph.

In the control flow graph fragment of Figure 1.5, suppose the goal is to execute the statement corresponding to node 3. Inputs which diverge away down the later true branch from node 2, are rewarded better objective values than those diverging away down the earlier true branch from node 1. The measure of closeness to the test goal with respect to control flow through the program is often referred to as the "approximation level" or "approach level". This measure can be combined with the "branch distance" measure. The branch distance measure rewards solutions that come close to evaluating branch conditions in the desired way. For example, the predicate at the decision at node 2 is `b > 0`. Inputs that are "closer" to evaluating this condition as false need to be rewarded better values than those further away. For example, inputs where `b = 2` should be rewarded better objective values that those where `b` is equal to 3, 4 or 5 and so on.

The objective function helps guide the search to the required test data. Such guidance is not a feature of random search. For random test data generation, input vectors for structures not normally executed by pure chance, i.e. those only executed by a small number of inputs in comparison to the overall input domain, may not be found. This is not to say, however, that search-based approaches will always find test data in all cases.

```
CFG
Node
 s    void f1(int a, int b)
      {
 1        if (a > 0)
              // ...
          else
 2            if (b > 0)
                  // ...
              else
 3                // target statement
 e    }
```

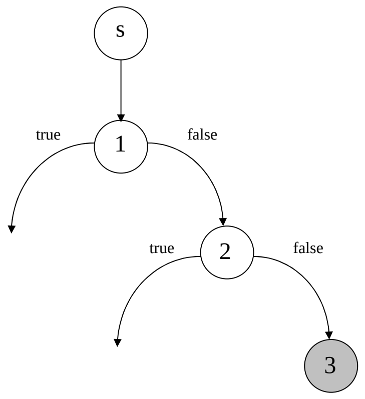


Figure 1.5: An example program and control flow graph (CFG) for applying test data search

1.2.1 The Problem of Internal Variables

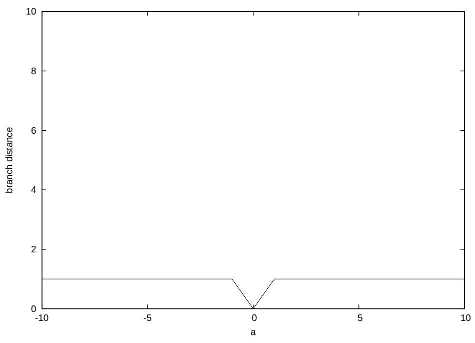
The use of internal variables in the conditions of programs can result in a degree of "information loss" when computing the branch distance measure, producing coarse or flat objective function landscapes for structures within the program. This in turn results in the search receiving less guidance to the required test data, making it difficult for the search algorithm to find the inputs that will evaluate conditions in the program in the desired way.

The degree of difficulty depends on the level of information lost, which in turn depends on the type of the internal variable and the form of assignments to it that appear in the program. Some internal variables may only result in a small amount of information loss, which may not affect the success of the search. However, in extreme cases, such as in the case of boolean "flag" variables, almost all useful branch distance information is lost. This is because the flag variable can only have one of two values - true or false, which in turn means the branch distance will also only have one of two values - one or zero. This causes plateaux in the objective function landscape - one plane corresponding to the "one" distance, or all input vectors that do not cause the target structure to be covered, and one plane corresponding to the "zero" distance, corresponding to the required test data. No guidance is provided to the search as to how to navigate from one plane in the objective function landscape to the other.

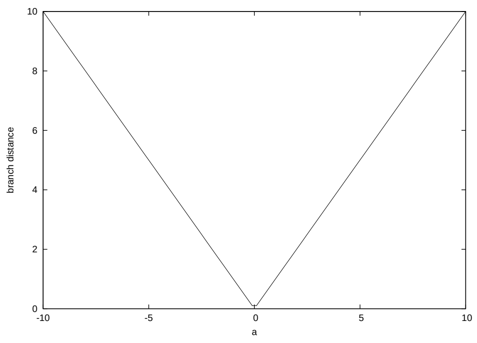
This is true in the example of Figure 1.6a. The plateaux corresponding to the "false" value of the flag can be seen clearly in Figure 1.6b. The flag is only true when the input value of `a` is zero, but the search receives no "direction" as to how to find this value. If the search compares a negative input of `a` with its

```
CFG
Node
 s    void f2(int a)
      {
 1        int flag = 0;

 2        if (a == 0)
 3            flag = 1;

 4        if (flag)
 5            // target statement
 e    }
```

(a)



(b)



(c)

Figure 1.6: Branch distances as a result of a flag variable. (a) Program code. The target statement is node 3 which requires the condition at node 2 to be evaluated as true. (b) Plot of true branch distance values from node 3, calculated using `|1 - flag|`. (c) Plot of potential distances using the original input value, i.e. `|0 - a|`.

neighbours, it does not know whether to move towards increased values (closer to the required zero value of `a`) or decreased values (further away from the required zero value of `a`), since the objective values returned for these inputs are exactly the same value. The search therefore becomes random. An exhaustive search could be employed, but if the input domain is very large, this could be impractical.

Figure 1.6c provides a contrasting landscape which does provide guidance to the required value of zero.

1.3 The Problem of this Thesis: States and Evolutionary Test Data Generation

To date, work in the field of search-based structural test data generation has largely focused on the testing of individual program functions with input-output behaviour. Test data is generated for atomic function calls.

However functions and components at higher system levels can store internal data, and can exhibit different behaviours based on the state of that data. This presents new challenges to the test data generation method. The first challenge is to generate a *sequence* of inputs to the test object, since certain program structures may require the test object to be in a particular state in order for them to be covered. For example, statements popping a value from a stack would not normally be covered unless the stack was in a non-empty state. The second challenge involves the problem of internal variables. State-based test objects by their very nature contain internal variables in order to manage their state. This can become problematic when internal variables like flags are used to manage or query the state, because the search may have difficulties in finding input sequences in order to cover certain structures within the program.

This thesis considers various possible solutions to the internal variable problem. One possible solution is to apply ideas from a technique known as the *chaining approach*. The basic idea of the chaining approach is to identify a sequence of statements that need to be executed prior to the target structure. These statements involve assignments to internal variables. By requiring they are executed, information previously unavailable to the search can be made use of, possibly guiding it into potentially promising, unexplored areas of the test object's input domain. In this way, the chances of finding input data to troublesome structural targets may be improved.

1.4 Aims and Objectives of this Thesis

The aims of this thesis encompass the following, general, objectives:

1. To identify the problems that test objects with state behaviour can cause for evolutionary structural test data generation for procedural programs written in the C language; and

2. To propose extensions to the evolutionary test data generation framework so that test data generation might be improved.

The last section briefly introduced the problems caused by states in test objects. This thesis aims to improve the evolutionary approach by allowing it to generate input sequences, as well as incorporating a chaining method for dealing with problems of internal variables. The finer, detailed, aims and objectives of this thesis are therefore as follows:

1. Initial hybridization of the evolutionary structural test data generation approach with the chaining approach;
2. Evaluation of this hybrid approach in the presence of problematic internal variables;
3. The extension of this hybrid approach to generate input sequences;
4. Evaluation of this extended hybrid approach for test objects with internal states and internal state variables.

1.5 Contributions of this Thesis

The contributions of this thesis are as follows:

1. The investigation and identification of the problems caused by test objects with state behaviour for the current, state of the art, evolutionary structural test data generation approach (referred to as simply the *standard evolutionary approach*);
2. The proposal of an encoding of individuals to allow the generation of input sequences involving multiple callable functions, as part of a method referred to as the *sequence evolutionary approach*;
3. The proposal of a method for integration of the chaining approach with evolutionary search, referred to as the *hybrid approach*;
4. Demonstration that the hybrid approach can improve on coverage levels for programs with input-output behaviour and internal variables, when compared with the standard approach;
5. The addressing of some limitations and weakness of the chaining algorithm so that further relevant event sequences can be found, and demonstration of the benefit of these improvements in a number of cases;
6. Extension of the hybrid approach so that input sequences can be generated, in order for generation of test data for test objects with state behaviour, referred to as the *sequence hybrid approach*;

7. Demonstration that the sequence hybrid approach improves coverage levels for test objects with state behaviour, when compared with the standard and sequence evolutionary approaches.

1.6 Overview of the Structure of this Thesis

This thesis is organized as follows:

Chapter 2 - Search-Based Software Test Data Generation surveys the literature in the field. The chapter begins by describing the various meta-heuristic search techniques used to date in automatically generating test data, including hill climbing, simulated annealing and evolutionary algorithms. The chapter then moves on to examine how these techniques have been used to automate test data generation for structural (white-box), functional (black-box), non-functional and grey-box test criteria.

Chapter 3 - The State Problem fully introduces and describes the state problem for evolutionary structural test data generation. Manifestations of the problem are seen in attempting to generate test data for a handful of test objects using the standard evolutionary approach. The chapter then reviews some specific work in the literature relating to evolutionary input sequence generation. A method for generating input sequences for test objects with a multiple number of callable functions (for example as part of a module) is proposed. Through a series of experiments, it is found that this sequence evolutionary approach still encounters difficulties with test objects containing internal variables.

Chapter 4 - Revisiting the Internal Variable Problem takes another look at the internal variable problem, with particular focus on test objects with internal states. Further works in the literature of relevance to the problem are evaluated. One of these works is the chaining approach. The basic idea of the chaining approach is to identify a sequence of statements that need to be executed prior to the target structure, and has potential to overcome the internal variable problem.

Chapter 5 - Hybridizing Evolutionary Testing with an Extended Chaining Approach discusses the method for hybridizing evolutionary test data generation with the chaining approach. Extensions are proposed to the chaining algorithm to address some weaknesses and limitations present in the method. Some experiments are performed using nine test objects with simple input-output behaviour and troublesome internal variables.

Chapter 6 - Extension of the Hybrid Approach for the State Problem extends the hybrid approach described in the previous chapter for test objects with state behaviour, and for the generation of input sequences. This technique is referred to as the “sequence hybrid approach”. Experiments are performed which demonstrate the superiority of the approach.

Chapter 7 - Conclusions and Future Work closes the main body of the thesis with concluding comments and proposals for future work.

The appendices are as follows:

Appendix A - Experimental Framework details the workbench used for conducting experiments that appear throughout the thesis.

Finally, **Appendix B - Program Code for Synthetic Test Objects with State Behaviour** archives the program source code for the state-based test objects used in experiments conducted in Chapters 3 and 6.

Chapter 2

Search-Based Software Test Data Generation: A Literature Review

2.1 Introduction

This chapter reviews work in the field of automatic test data generation through the use of metaheuristic search techniques, beginning with an introduction of each search technique used in automatic test data generation to date. This is followed by the application of the techniques to various testing types, starting with structural testing. Here, several basic concepts are introduced that are key to the remainder of this thesis. The discussion then moves on to the application of search techniques to other forms of testing, namely black-box (functional) testing, grey-box testing and non-functional testing.

All examples are presented in the C language [KR88].

2.2 Metaheuristic Search Techniques

Metaheuristic search techniques are high-level frameworks which use heuristics to find solutions to problems without the need to perform a full exhaustive enumeration of a search space. They are therefore well suited to finding good solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical. These frameworks are not standalone algorithms in their own right, but rather "strategies" that are ready for adaptation to specific problems. Key decisions in adapting a metaheuristic search strategy include the definition of an encoding for candidate solutions, so that they can be manipulated by the search. A good encoding will ensure that candidate solutions sharing a number of similar properties will be "neighbours" in encoded solution space. Another key decision is the definition of a problem-specific objective function, which the search uses as a guide to the quality of candidate solutions.

Chapter 1 briefly introduced some metaheuristic techniques that have been used in software test data generation, namely hill climbing, simulated annealing and evolutionary algorithms. This section discusses these methods in more detail. Further treatment can be found in reference [Ree95].

The last decade has seen the emergence of many new techniques, which have not been exploited by the test data generation techniques presented here. The interested reader is directed to reference [CDG99], which gives treatment to some of these methods - for example ant colony algorithms, scatter search and memetic algorithms.

2.2.1 Hill Climbing

"Hill climbing" is a well known local search algorithm. Hill climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighbourhood of this solution is investigated. If a better solution is found, then this replaces the current solution. The neighbourhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again, and so on, until no improved neighbours can be found for the current solution.

In a "steepest ascent" climbing strategy, all neighbours are evaluated, with the neighbour offering the greatest improvement chosen to replace the current solution. In a "random ascent" strategy (sometimes referred to as "first ascent"), neighbours are examined at random and the first neighbour to offer an improvement is chosen. A high level description of the algorithm can be seen in Figure 2.1.

Hill climbing is simple and gives fast results. However it is easy for the search to yield sub-optimal results when the hill climbed leads to a solution that is locally optimal, but not globally optimal. In such cases, the search becomes trapped at the peak of a hill, unable to explore other areas of the search space. The search will also become stuck along plateaux in the landscape. In such circumstances, no neighbouring solution is deemed to offer an improvement over the current solution, since they all have the same objective value. Therefore, in non-trivial landscapes, results obtained with hill climbing are highly dependent

```
Select a starting solution $s \in S$
Repeat
    Select $s' \in N(s)$ such that $obj(s') > obj(s)$ according to ascent strategy
    $s \leftarrow s'$
Until $obj(s) \geq obj(s'), \forall s' \in N(s)$
```

Figure 2.1: High level description of a hill climbing algorithm, for a problem with solution space S; neighbourhood structure N; and obj, the objective function to be maximized

on the starting solution. A common extension to this algorithm is to incorporate a series of "restarts" involving different initial solutions, to sample more of the search space and minimise this problem as much as possible.

2.2.2 Simulated Annealing

It is desirable to have a search framework that is less dependent on the starting solution. Simulated annealing is similar in principle to hill climbing. However, by probabilistically accepting poorer solutions, simulated annealing allows for less restricted movement around the search space. The probability of acceptance p of an inferior solution changes as the search progresses, and is calculated as:

$$p = e^{-\frac{\delta}{t}}$$

where δ is the difference in objective value between the current solution and the neighbouring inferior solution being considered, and t is a control parameter known as the *temperature*. The temperature is cooled according to a *cooling schedule*. Initially the temperature is high, in order to allow free movement around the search space, and so that dependency on the starting solution is lost. As the search progresses, the temperature decreases. However, if cooling is too rapid, not enough of the search space will be explored, and the chances of the search becoming stuck in local optima are increased. The basic algorithm, for minimising an objective function, can be seen in Figure 2.2.

The name "simulated annealing" originates from the analogy of the technique with the chemical process of annealing - the cooling of a material in a heat bath. If a solid material is heated past its melting point, and then cooled back into a solid state, the structural properties of the cooled solid depend on the rate of cooling. An algorithm proposed by Metropolis *et al.* [MRR$^+$53] simulates the change in energy of the system when subjected to a cooling process, until it converges into a steady state. This algorithm was later proposed

```
Select a starting solution $s \in S$
Select an initial temperature $t > 0$
Repeat
    $it \leftarrow 0$
    Repeat
        Select $s' \in N(s)$ at random
        $\Delta e \leftarrow obj(s') - obj(s)$
        If $\Delta e < 0$
            $s \leftarrow s'$
        Else
            Generate random number $r$, $0 \leq r < 1$
            If $r < e^{-\frac{\delta}{t}}$ Then $s \leftarrow s'$
        End If
        $it \leftarrow it + 1$
    Until $it = num\_solns$
    Decrease $t$ according to cooling schedule
Until Stopping Condition Reached
```

Figure 2.2: High level description of a simulated annealing algorithm, for a problem with solution space S; neighbourhood structure N; num_solns, the number of solutions to consider at each temperature level t; and obj, the objective function to be minimized

as the basis of the search mechanism by Kirkpatrick *et al.* [KGV83].

2.2.3 Evolutionary Algorithms

Evolutionary algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection.

Genetic algorithms are probably the most well known form of evolutionary algorithm, having been conceived by John Holland in the United States during the late sixties. Genetic algorithms are closely related to evolution strategies, which were developed independently at the about the same time in Germany by Ingo Rechenburg and Hans-Paul Schwefel. For genetic algorithms, the search is primarily driven by the use of recombination - a mechanism of exchange of information between solutions to "breed" new ones - whereas evolution strategies principally use mutation - a process of randomly modifying solutions. Although these different approaches were developed independently, and with different directions in mind, recent work has incorporated ideas from both traditions - narrowing the differences between the two. The discussion here, however, focuses on genetic algorithms. For more information on evolution strategies, see references [BHS91, Bac96, Whi01].

Genetic Algorithms

The name "genetic algorithm" comes from the analogy between the encoding of candidate solutions as a sequence of simple components, and the genetic structure of a chromosome. Continuing with this analogy, solutions are often referred to as *individuals* or *chromosomes*. The components of the solution are sometimes referred to as *genes*, with the possible values for each component called *alleles*, and their position in the sequence the *locus*. Furthermore, the actual encoded structure of the solution for manipulation by the genetic algorithm is called the *genotype*, with the decoded structure known as the *phenotype*. For many applications, the genotype is simply a string of binary digits (this issue will be revisited in the context of test data generation). For example, a vector of three integers <`112`, `255`, `52`> in the range [`0`, `255`] might be represented as <`01110000`, `11111111`, `00110100`>. For real values, a decision must made on the precision to be used and what mapping should be used to the binary strings. One possibility, for example, is to scale real values onto integer values according to the required precision, and then use an integer encoding.

Genetic algorithms maintain a population of solutions rather than just one current solution. Therefore, the search is afforded many starting points, and the chance to sample more of the search space than local searches. The population is iteratively recombined and mutated to evolve successive populations, known as *generations*.

The recombination operator takes two parent solutions and "breeds" them to produce two new offspring. In one-point recombination, a single crossover point is chosen at random. A recombination of two individuals <`0`, `255`, `0`> and <`255`, `0`, `255`>, `000000001111111100000000` and `111111110000000011111111` in encoded form, with a single-point crossover chosen to take place at locus 12, would take place as follows:

$$\begin{array}{l|l} \texttt{000000001111} & \texttt{111100000000} \\ \texttt{111111110000} & \texttt{000011111111} \end{array} \rightarrowtail \begin{array}{l} \texttt{000000001111000011111111} \\ \texttt{111111110000111100000000} \end{array}$$

This produces two offspring - <`0`, `240`, `255`> and <`255`, `15`, `0`>. Multiple point crossover operators choose a fixed number of loci for recombination. Discrete recombination [MSV93], on the other hand, produces offspring where every component value is chosen randomly from one of the parents with equal probability.

Various selection mechanisms can be used to decide which individuals should be used to create offspring for the next generation. Key to this is the concept of the "fitness" of individuals. The fitness of an individual can be the value obtained directly from the objective function, or this value scaled in some way. The idea of selection is to favour the fitter individuals, in the hope of breeding fitter offspring. However, too strong a bias towards the best individuals will result in their dominance of future generations, thus reducing diversity and increasing the chance of premature convergence on one area of the search space. Conversely, too weak a strategy will result in too much exploration, and not enough evolution for the search to make substantial progress.

Holland's original genetic algorithm [Hol75] used *fitness-proportionate selection*. In this selection mechanism, the expected number of times an individual is selected for reproduction is proportionate to the individual's fitness in comparison with the rest of the population. The process is analogous to the use of a roulette wheel. Each individual is allocated a slice of the wheel in proportion to its fitness. The wheel is then spun N times in order to pick N parents. At the end of each spin, the position of the wheel marker denotes an individual selected to be a parent for the next generation. Fitness-proportionate selection has difficulties in maintaining a constant *selective pressure* throughout the search. Selective pressure is the probability of the best individual being selected, compared to the average probability of selection of all individuals. In the first few generations of the search, fitness variance is usually high. With fitness-proportionate selection, selective pressure will also be high, since the most highly-fit individuals will be granted the greatest opportunities to become parents. This can lead to premature convergence. Also in later generations, when fitness values amongst individuals are similar and the fitness variance of the population is correspondingly low, selective pressure is also low. This can lead to stagnation of the search.

Linear ranking of individuals is a technique which proposes to circumvent this problem. Individuals are sorted by fitness, with selection performed according to rank, rather than through the direct use of fitness values. A linear ranking mechanism with bias Z, where $1 < Z \leq 2$, allocates a selective bias of Z to the top individual, a bias of 1.0 to the median individual, and $2 - Z$ to the bottom individual. With a constant bias applied throughout the search, selective pressure is more constant and controlled [Whi89].

Tournament selection [DG91] is a noisy but fast rank selection algorithm. The population does not need to be sorted into fitness order. Two individuals are chosen at random from the population. A random number, $0 < r \leq 1$, is then chosen. If r is less than p (where p is the probability of the better individual being selected), the fitter of the two individuals 'wins' and is chosen to be a parent, otherwise the less fit individual is chosen. The competing individuals are returned to the population for further possible selection. This is repeated N times until the required number of parents have been selected. In all probability, every individual is sampled twice, with the best individual selected for reproduction twice, the median individual once, with the worst individual remaining unselected. The resulting selective bias is dependent on p. If $p = 1$, then in all probability a ranking with a bias of 2.0 towards the best individual is produced. If $0.5 < p \leq 1$, then the bias is less than 2.0.

Once the set of parents has been selected, recombination can take place to form the next generation. Crossover is applied to individuals selected at random with a probability p_c (referred to as the *crossover rate* or *crossover probability*). If crossover takes place, the offspring are inserted into the new population. If crossover does not take place, the parents are simply copied into the new population. After recombination, a stage of mutation is employed, which is responsible for introducing or reintroducing genetic material into the search, in the interests of maintaining diversification. This is usually achieved by flipping bits of the binary strings at some low probability rate p_m, which is usually less than 0.01.

A high-level description of a genetic algorithm can be seen in Figure 2.3. The initial population is generated at random, or *seeded* with pre-set individuals. The search is terminated when some stopping criterion has been met, for example when the number of generations has reached some pre-imposed limit.

Advanced Encodings and Operators

Traditionally chromosomes are represented as a string of binary digits. A problem with standard binary encoding is the disparity that can occur between solutions that are close to each other in unencoded solution space, but are far apart in the encoded binary representation. For example in a standard binary encoding the integer `7` is represented as `0111`, yet `8` is represented as `1000`. Therefore, the crossover and mutation operators must change all four bits to move from one integer value to the neighbouring other. An alternative is the use of a gray code. A gray code is a binary representation where adjacent integers are also hamming distance 1 neighbours in hamming space. For example, in *standard binary reflected gray code*, `7` is represented as `0100`, and `8` as `1100`. Empirical evidence has shown that gray codes are generally superior to standard binary encodings [Whi99, WRDM96].

Goldberg argues that binary representation decomposes the chromosome into the largest number of smallest possible building blocks in order for the recombination and mutation operators to work most effectively [Gol89]. However, this is disputed by Antonisse [Ant89], who advocates the use of more expressive alphabets. Davis [Dav96] supports this view. For nine real-world applications using genetic algorithms over a variety of problem domains, Davis found that real-valued representations always outperformed binary encodings (real-valued encodings are also the representational choice of evolution strategies [Whi01]). Of course, the use of a real-valued encoding raises the question of how recombination and mutation should work. The recombination operator only requires an underlying sequence representation, and as such can operate as for binary encodings. Possibilities for the mutation operator include the replacement of a real number in the chromosome with a new, randomly generated number. More advanced mutation operators are based on *real number creep*. These operators sweep across the chromosome, pushing values up and down by a small amount. In this way, an element of local search is incorporated [Dav96].

Competition and Migration across Subpopulations

Splitting the overall population into subpopulations is one way to seek improvement to the basic genetic algorithm. Since each initial population will sample the search space in different ways, each population will explore a different search trajectory. Thus diversity is encouraged. Subpopulations can compete for resources [SVM94, SVM96]. In this model, successful subpopulations are allowed to proliferate, receiving a greater proportion of individuals, whereas weaker subpopulations are punished, with individuals dying off. By employing *Migration* [Tan89, SWM91, Gor91], subpopulations can occasionally exchange a small number of solutions with one another. This trading of genetic material helps to prevent premature convergence in one population.

The subpopulation model, more commonly referred to as the *island model* or *course-grained model*, is a natural way to parallelize the search. However the model generally yields improvement even when run on a single processor [Whi01].

Genetic algorithms have been successfully applied to a wide range of problems. For introductory texts, see references [Gol89, Mit96]. For shorter overviews and tutorials, see references [SP94, Whi01, Whi94].

2.3 Structural (White-Box) Testing

Structural or white-box testing is the process of deriving tests from the internal structure of the software under test. This section summarizes some of the achievements in automating structural test data generation through the use of

```
Randomly generate or seed initial population $P$
Repeat
    Evaluate fitness of each individual in $P$
    Select parents from $P$ according to selection mechanism
    Recombine parents to form new offspring
    Construct new population $P'$ from parents and offspring
    Mutate $P'$
    $P \leftarrow P'$
Until Stopping Condition Reached
```

Figure 2.3: High level description of a genetic algorithm

metaheuristic techniques. These are compared with earlier related approaches. Before this, some basic concepts are reviewed.

2.3.1 Basic Concepts

Many forms of structural testing make reference to the *control flow graph* (CFG) of the program in question. A control flow graph for a program F is a directed graph $G = (N, E, s, e)$, where N is a set of nodes, E is a set of edges, and s and e are respective unique entry and exit nodes to the graph. Each node $n \in N$ is a statement in the program, with each edge, $e = (n_i, n_j) \in E$, representing a transfer of control from node n_i to node n_j. An example of a control flow graph can be seen for a version of a triangle classification program in Figure 2.4. The triangle classification program is a benchmark used in many testing papers. Assuming three non-zero, non-negative integer lengths for the sides of a triangle, the program decides if the triangle is isosceles, scalene, equilateral, or invalid. Nodes corresponding to decision statements (for an example an `if` or a `while` statement) are referred to as *branching nodes*. In the triangle example, branching nodes are nodes 1, 5, 9, 13, 16 and 18. Outgoing edges from these nodes are referred to as *branches*. The condition determining whether a branch is taken is referred to as the *branch predicate*. For the true branch from node 1, the branch predicate is `a > b`.

An *input vector* I is a vector $I = (x_1, x_2, \ldots, x_k)$ of input variables to the program F. The domain of an input variable x_i, $1 \leq i \leq k$, is the set if all values that x_i can take on. The *domain* of the program F is the cross product $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_k}$ where each D_{x_i} is the domain for the input variable x_i. A *program input* $\mathbf{x}$ is a single point in the k-dimensional input space D, $\mathbf{x} \in D$.

A path P through a control flow graph is a sequence $P =< n_1, n_2, \ldots, n_m >$, such that for all $i, 1 \leq i < m$, $(n_i, n_{i+1}) \in E$. A path is said to be *feasible* if there exists a program input for which the path is traversed, otherwise the path is said to be *infeasible*.

A *definition* of a variable v is a node which modifies the value of v, for example an assignment statement or an input statement. The variable `type` is defined in the triangle program at node 14. A *use* of a variable v is a node in which v is referenced, for example in an assignment statement, an output statement, or a branch predicate expression. In the triangle classification example, the variables `a` and `b` are used at node 1.

A *definition-clear path* with respect to variable v is a path within which v is not modified. In the triangle example, all paths from node 13 are definition-clear with respect to variables `a`, `b` and `c`. However, no path from node 13 is definition clear with respect to `type`.

The term *control dependency* is used to describe the reliance of a node's execution on the outcome at previous branching nodes [FOW87]. A node z is *post-dominated* by a node y in G if and only if every path from y to the exit node e contains z. Node z post-dominates a branch (y, x) if and only if every path from y to the exit node e through (y, x) contains z. The node z is *control dependent* on y if and only if z post-dominates one of the branches of y, and z does not post-dominate y. In the triangle example, node 17 is control dependent on node 16, which in turn is control dependent on node 13. Node 13 itself has no control dependencies, other than that of the external condition, *entry*, that causes the procedure to be executed. This information can be captured by a control dependence graph. Figure 2.5 shows the control dependence graph for the triangle program.

The techniques now described have been implemented for experimentation with a variety of programming languages. For consistency, however, all examples here are presented in C.

2.3.2 Static Structural Test Data Generation

Static structural test data generation is based on analysis of the internal structure of the program, without requiring that the program is actually executed.

Symbolic Execution

Symbolic execution [Kin75, Kin76] is not the execution of a program in its true sense, but rather the process of assigning expressions to program variables as a path is followed through the code structure. The technique can be used to

CFG Node	
s	`int tri_type(int a, int b, int c)`
	`{`
	`    int type;`
1	`    if (a > b)`
2-4	`    {   int t = a; a = b; b = t;   }`
5	`    if (a > c)`
6-8	`    {   int t = a; a = c; c = t;   }`
9	`    if (b > c)`
10-12	`    {   int t = b; b = c; c = t;   }`
13	`    if (a + b <= c)`
	`    {`
14	`        type = NOT_A_TRIANGLE;`
	`    }`
	`    else`
	`    {`
15	`        type = SCALENE;`
16	`        if (a == b && b == c)`
	`        {`
17	`            type = EQUILATERAL;`
	`        }`
18	`        else if (a == b || b == c)`
	`        {`
19	`            type = ISOSCELES;`
	`        }`
	`    }`
e	`    return type;`
	`}`

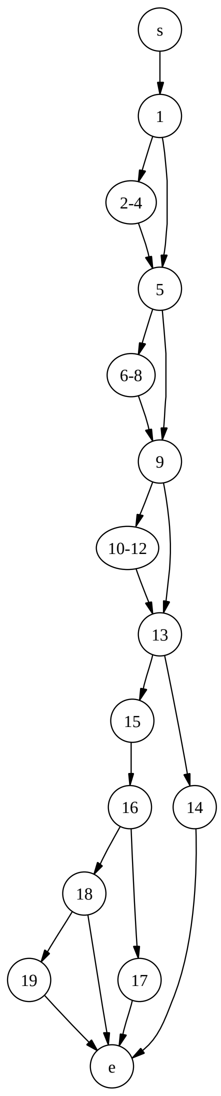


Figure 2.4: A triangle classification program and its corresponding control flow graph

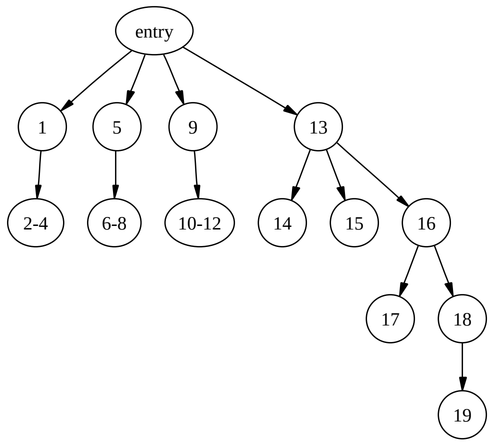


Figure 2.5: Control dependence graph for the triangle classification program of Figure 2.4

derive a constraint system in terms of the input variables which describes the conditions necessary for the traversal of a given path [Cla76, BEL75, RHC76].

A forward traversal (or forward substitution) of a path, can be demonstrated with the triangle classification program in Figure 2.4. Suppose the path $< s$, 1, 5, 9, 10, 11, 12, 13, 14, $e >$ is to be executed. The input variables `a`, `b` and `c` are assigned the constant variables `i`, `j` and `k` respectively. At nodes 1 and 5, the respective false branches are to be taken. Therefore, the first and second constraints of the constraint system for this path are:

(1) `i <= j`
(2) `i <= k`

The path also requires that the true branch be taken from node 9. This requires the addition of a third constraint:

(3) `j > k`

The following expressions are assigned at nodes 10 through to 12 respectively:

```
t = j
b = k
c = t
```

A fourth and final constraint from node 13 then needs to be added. With `a = i`, `b` now equal to `k`, and `c = t = j`, this becomes:

(4) `i + k <= j`

Backward path traversal is also possible, starting with the final node and following the path in a reverse manner to the start node. The resulting constraint system is the same as for forward traversal, but no storage is required for the intermediate symbolic expressions of variables. Forward traversal, however, allows for early detection of infeasible paths if the constraints generated are inconsistent. Consider the path $< s, 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, \ldots, e >$ which requires that the true branches are taken from nodes 1 and 5, and that the false branch from node 9 is taken. The constraints derived from the branching predicates from the initial section of the path through to node 9 are:

```
(1)  i > j
(2)  j > k
(3)  i <= j
```

Clearly constraints 1 and 3 are contradictory, indicating that the path is infeasible. Backward traversal would have meant symbolic execution of the path backwards from e through to 13 first, and then backwards through the nodes to node 1 before it would be possible to determine this fact.

Solutions to the constraint system are input data which will execute the path. Constraint satisfaction problems are in general NP-complete [GJ79]. However, if the constraints are linear, linear programming techniques can be applied [Cla76]. Heuristic methods can be used to attempt the finding of a solution where this is not the case. For example Boyer *et al.* [BEL75] employ hill climbing. Ramamoorthy *et al.* [RHC76] use a trial and error procedure, monitoring the effects of random-value assignments to variables in the constraint system. It is unlikely, however, that this procedure would be efficient for non-trivial programs.

If the test goal is the execution of a particular statement, all paths leading to the statement are explored. This is a problem in the presence of loops, due to the potential number of paths that may need to be examined. In Clarke's test data generator system [Cla76], a path has to be manually selected by the tester. Many generators symbolically simply execute the loop K times, where K is specified by the tester or chosen by the system [RHC76]. A large number of constraints generated using this method, however, are not satisfiable.

Symbolic execution has several other problems, for example resolving computed storage locations such as array subscripts.

```
a[i] = 0;
a[j] = 1;
if (a[i] > 0)
{
    // perform some action
}
```

In the above code fragment, it is not known in general whether `a[i]` and `a[j]` refer to the same element, because the variables `i` and `j` are not bound to specific values. This information is important, since if `i` and `j` are equal, then the value of `a[i]` in the condition is `1` and the branch predicate evaluates to true. If not, the value of `a[i]` is `0` and the predicate evaluates to false. Boyer *et al.* [BEL75] and Ramamoorthy *et al.* [RHC76] suggest possible solutions to this problem. Both methods significantly increase the complexity and memory requirements of the symbolic execution system. A similar problem occurs with the use of pointers. In the following example, it is not known if `a` and `b` refer to the same location. Without this knowledge, the expression to assign to `c` cannot be determined.

```
*a = 0;
*b = 1;
c = *a;
```

Further difficulties include the handling of procedure calls. A common solution is to simply inline the called procedure into the calling routine [RHC76]. However the number of paths can grow very rapidly with this approach.

Although any computable function can be written without the use of arrays, pointers or procedure calls, it is not normal practice for programmers to avoid such constructs simply because of the flexibility they offer, and the role they play in reducing the complexity of program code.

Domain Reduction

Domain reduction is a test data generation technique that was originally employed as part of constraint-based testing, developed by DeMillo and Offutt [DO91]. Constraint-based testing builds up constraint systems which describe the given test goal. The solution to this constraint system brings about satisfaction of the goal. The original purpose of Constraint-based Testing was to generate test data for mutation testing. *Reachability constraints* within the constraint system describe conditions under which a particular statement will be reached. *Necessity constraints* describe the conditions under which a mutant will be killed. Symbolic execution is used to develop the constraints in terms of the input variables. *Domain reduction* is then used to attempt a solution to the constraints. This procedure begins with the domains of each input variable. These can be derived from type or specification information, or be supplied by the tester. The domains are then reduced using information in the constraints, beginning with those involving a relation operator, a variable and a constant, and constraints involving a relation operator and two variables. Remaining constraints are then simplified by back-substituting values. When no further simplification is possible, the input variable with the smallest remaining domain is chosen, and a random value is assigned to it. The value of this variable is then back-substituted throughout the constraint system, in order to allow further reduction of the domains of remaining variables. If all variables can be assigned values in this manner, then the constraint system will have been satisfied; otherwise the variable assignment stage is repeated, in the hope of this time successfully selecting appropriate random numbers for the variables.

With constraint-based testing, constraints must be computed before they are analyzed. Since these constraints are derived using symbolic execution, the method suffers from similar problems involving loops, procedure calls and computed storage locations. Dynamic domain reduction was introduced by Offutt *et al.* [OJP99] with the intent of addressing some of these issues. Although called *dynamic* domain reduction, the technique still has the characteristic that the program is not executed with real input values. As with standard domain reduction, dynamic domain reduction starts with the domains of the input variables. However, in contrast to standard domain reduction, these domains are reduced "dynamically" during the symbolic execution stage, using constraints composed from branch predicates encountered as the path is followed. If the branch predicate involves a variable comparison, the domains of the input variables responsible for the outcome at the decision are split at some arbitrary "split point", rather than assigning random input values. For example if the initial domains of two input variables `y` and `z` are [-10...10] and a branch predicate `y < z` is encountered which needs to be executed as true, the domains might be split leaving the domain of y to be [-10...0] and z to be [1...10]. A back tracking procedure can be used to correct any spurious split points if the execution can only proceed so far down the specified path, and is unable to continue further due to a bad decision made earlier in the reduction process.

Despite setting out to deal with problems traditionally encountered by techniques based on symbolic execution, dynamic domain reduction still suffers with difficulties due to computed storage locations and loops. Furthermore, it is not clear how domain reduction techniques handle non-ordinal variable types, such as enumerations.

2.3.3 Dynamic Structural Test Data Generation

As has already been discussed, the relationship between input data and internal variables for structural test data generation is difficult to analyse statically in the presence of loops and computed storage locations. *Dynamic methods* execute the program in question with some input, and then simply observe the results via some form of program instrumentation. Since array subscripts and pointer values are known at run-time, many of the problems associated with symbolic execution can be circumvented.

Random Testing

Random testing simply executes the program with random inputs and then observes the program structures executed. This technique works well for simple programs. However structures that are only executed with a low probability are often not covered. Consider the triangle classification example once more (Figure 2.4). The true branch from node 16 requires that the three input values for `a`, `b` and `c` are all equal. Such a branch is unlikely to be executed by chance. Even if the domain of integer values for each variable were limited to values between 1 and 100, the probability of all three variables being selected with the same value is 1 in 10,000. In such cases a more directed search technique is required to locate test data.

Applying Local Search

Miller and Spooner [MS76] were the first to combine the results of actual executions of the program with a search technique. Their method was originally designed for the generation of floating-point test data, however the principles are more widely applicable. The tester selects a path through the program, and then produces a straight-line version of it, containing only that path. Branching statements are then replaced with a "path constraint" of the form $c_i = 0$; $c_i > 0$; or $c_i \geq 0$; where c_i is an estimate of how close the constraint is to being satisfied. For example, a branch predicate of the form `a == b` might be rearranged into the path constraint $abs(a - b) = 0$. Take the triangle example and the execution of the path $< s$, 1, 5, 9, 10, 11, 12, 13, 14, $e >$ again. The straight-line program with its respective path constraints would be re-arranged as follows:

```
int tri_type(int a, int b, int c)
{
    int type;
    (c1 = (b − a)) >= 0
    int t = a; a = b; b = t;
    (c2 = (c − a)) >= 0
    (c3 = (b − c)) > 0
    (c4 = (c − (a + b))) >= 0
    type = NOT_A_TRIANGLE;
}
```

Note that the value of c_2, c_3 and c_4 are dependent on the computations between c_1 and c_2. However, this information is not required for the derivation of the path constraints, as it would be for the process of test data generation using symbolic execution.

Using these constraints, a function f is constructed. The value of f provides a real-valued estimate of how close all of the constraints are to being satisfied, being negative when one or more of the constraints remains unsatisfied, and positive when all of the constraints are satisfied. Input values of `a`, `b` and `c` are then sought through the use of numerical maximisation techniques, which attempt to push the value of f closer and closer to zero, in the hope of eventually making it positive.

Under normal conditions, execution of the complete path is not possible until branch predicates encountered along the path are evaluated in the required manner. However, in the straight-line version of the program, it is possible for run-time errors to occur which would not have been possible in the original program. In the following segment of code, if execution is allowed to proceed down the true branch with values of `i` less than zero, or greater than `size`, an error will be induced, because the array index used in the assignment statement will be out of bounds:

```
if (i >= 0 && i < size)
{
    a[i] = 0;
}
```

It was not until 1990 that the ideas of Miller and Spooner were extended by Korel [Kor90] for Pascal programs. In this work, the test data generation procedure worked on an instrumented version of the original program without the need for a straight-line version to be produced. The search targeted the satisfaction of each branch predicate along the path in turn, circumventing issues encountered by the work of Miller and Spooner. To execute some desired path, the program is initially executed with some arbitrary input. If during execution an undesired branch is taken - one which deviates from the desired path - a local search for program inputs is invoked, using an objective function derived from the predicate of the desired, alternative branch. This objective function describes how "close" the predicate is to being true. The value obtained is referred to as the *branch distance*.

Take the triangle example and the execution of the path $< s$, 1, 5, 9, 10, 11, 12, 13, 14, $e >$ again. If the function is executed with the program input `(a=10, b=20, c=30)`, control flow successfully follows the false branches from nodes 1 and 5. However control flow diverges away from the intended path down the false branch at node 9. At this point the local search is invoked to change the program inputs so that the alternative true branch is taken. If, in general, the branch predicate is assumed to be of the form $a\ op\ b$, where a and b are arithmetic expressions and op is a relational operator, an objective function of the form $f\ rel\ 0$ is derived, where f and rel are given in Table 2.1. The function is to be minimized, being positive (or zero if rel is '$<$') when the current branch predicate for the required branch is false, and negative (or zero if rel is '$=$' or '$\leq$') when it is true. For the predicate of the true branch from node 9, the objective function is `c - b > 0`. The value of this function for the program input `(a=10, b=20, c=30)` is `30 - 20 = 10`. The program must be instrumented so that objective values can be computed. This can be performed within the branching expression, for example as follows:

```
if (eval_obj(9, b, c))
{
    ...
```

Here, the program function `eval_obj` reports branch distances at node 9 using the local values of `b` and `c`. This function will then return a boolean value corresponding to the evaluation of the original branching expression, in order for program execution to resume as normal.

The local search for deriving input values in accordance with the objective function is known as the *alternating variable* method. Each input variable is taken in turn and its value adjusted, keeping the other variable values constant. The first stage of manipulating an input variable is called the *exploratory* phase. This probes the neighbourhood of the variable by increasing and decreasing its original value. If either move leads to an improved objective value, a *pattern* phase is entered. In the pattern phase, a larger move is made in the direction

Table 2.1: Korel's objective functions for relational predicates

Relational predicate	f	rel
$a > b$	$b - a$	$<$
$a \geq b$	$b - a$	$\leq$
$a < b$	$a - b$	$<$
$a \leq b$	$a - b$	$\leq$
$a = b$	$abs(a - b)$	$=$
$a \neq b$	$-abs(a - b)$	$<$

of the improvement. A series of similar moves is made until a minimum for the objective function is found for the variable. The next input variable is then selected for an exploratory phase.

Return to the triangle example again, for which execution had diverged from the intended path at node 9. Decreases and increases of `a` have no effect on the objective value. Therefore `b` is chosen. A decrease of `b` leads to a worse objective value, but an increase leads to an improvement. The pattern phase is entered for `b`, which will be increased until `b > c`. Suppose the value `31` is reached. The new input vector is now `(a=10, b=31, c=30)`. Control flow now proceeds through branching node 9 as desired, however execution now diverges away at node 13, since the value of `a + b` at the node is greater than the value of `c`. The local search is invoked again, this time to adjust the input values so that the true branch is taken from node 13, whilst maintaining the already correct sub-path up to this node. The new objective function, derived from the true branch predicate, is `(a + b) - c <= 0`. A decrease of the input value of `b` leads to a violation of the sub-path up to node 9, yet an improved value of the objective function is found for an increase of `b` (since the internal values of `b` and `c` are swapped at nodes 10-12). Eventually the input vector `(a=10, b=40, c=30)` will be found. This input vector evaluates branching node 13 as true, and the complete path is executed.

As with all local searches, the final result is dependent on the starting solution. Consider the example of Figure 2.6. If the input is initially selected as `(a=10, b=10, c=10)`, control flow proceeds directly down to the final branching node. However the variable `c` cannot be changed to a value less than `0`, because the already successful sub-path up to the final branching node will be violated. In this case, the search will fail.

Heuristic search methods have the potential to make moves through variable values that cannot lead to an improvement in the value of the current cost function. This can lead to many wasteful and costly executions of the program.

```
void nested_example(int a, int b, int c)
{
    if (a == b)
        if (b == c)
            if (c < 0)
                // target
}
```

Figure 2.6: Example program with nested structures

In the triangle example, changing the value of the input variable `c` does not have an effect on branching node 1. In order to make the search more efficient, Korel's work makes use of extra information derived from the program, in the form of an "influences" graph. An influences graph is used to detect which input variables are able to influence the outcome at the current branching node, as determined using dynamic data flow analysis. A risk analysis of input variables is also undertaken in order to decide if they could potentially violate the already successful sub-path. For example at node 5, it is more attractive to manipulate `c` rather than `a` or `b`, since changing `a` or `b` may change the current successful sub-path through node 1.

Gallagher and Narasimhan [GN97] built on Korel's work for programs written in Ada. In particular, this was the first work to record support for the use of logical connectives within branch predicates. For predicates of the form A *and* B, the overall objective value is formed from the summation of the individual objective values of the expressions A and B. For predicates of the form A *or* B, the objective value is the minimum value of the individual objective values of the expressions.

The Goal-Oriented Approach

In his paper published in 1992, Korel developed what became known as the goal-oriented approach [Kor92]. All of the techniques concentrate on the execution of a path. For fulfilling a structural coverage criterion like statement coverage, this means a path has to be selected for each individual uncovered statement. The goal-oriented approach removes this requirement. This is achieved through the classification of branches in the control flow graph of the program with respect to a target node as either *critical*, *semi-critical* or *non-essential*. This can be performed automatically on the basis of the program's control flow graph.

For branches leaving a node on which the target is control dependent, a *critical branch* is the edge which leads the execution path away from the target node. If control flow is driven down a critical branch, there is no prospect of the target being reached. Therefore, an objective function, of the form outlined in the previous section, is associated with the branch predicate of the alternative branch. The alternating variable search method is then employed to seek inputs so the alternative branch is taken instead. If the required inputs cannot be found, the overall process terminates, with the target remaining unexecuted.

A *semi-critical* branch is one which leads to the target node, but only via the backward edge of a loop. The alternative branch from the same branching node leads directly to the target node. In the case where the execution is driven down a semi-critical branch, the alternating variable method is again invoked to seek inputs for the execution of the alternative branch. If suitable input values cannot be found, however, the process does not terminate. Execution is allowed to flow down the semi-critical branch, in the hope of taking the alternative branch in the next iteration of the loop.

Finally, a *non-essential* branch is neither critical or semi-critical. Non-essential branches do not determine whether the target will be reached, regardless of their position in the control flow graph. Therefore, execution is allowed to proceed unhindered through these branches.

Take the example of Figure 2.7, with the target being the execution of node 5. The classification of each branch can be seen from the control flow graph in Figure 2.8. The false branches from nodes 1 and 3 are critical since node 5 cannot be reached if they are executed. The false branch from condition 4 is semi-critical, because although control flow diverges away from the target at this point, the target may still be reached in the next iteration of the loop. If the input vector is `(a=0)` the false branch from condition 1 is taken, and so the search procedure is invoked to change the value of `a`. Control flow proceeds through down the true branch from node 1, but from node 4 the false branch is taken. However, the search cannot change the outcome at this branch, and so the flow of control is allowed to continue around the loop a further nine times upon which the true branch from node 4 is taken, and the target is reached.

As the goal-oriented method also employs the alternating variable local search, it suffers from similar problems to those of Korel's original approach. The removal of the requirement to select a path, although relieving some effort on behalf of the tester, introduces new ways in which the test data search can fail. Take the example of Figure 2.9 and the execution of the true branch from node 4. The true branch is only taken for objective values less than or equal to zero. Consider what happens when the initial input vector is selected so that `a` is less than zero (approximately half of the input domain). With such

```
CFG
Node
 s    void goal_oriented_example(int a)
      {
 1        if (a > 0)
          {
 2            int b = 10;
 3            while (b > 0)
              {
 4                if (b == 1)
                  {
 5                    // target
                  }

 6                b --;
              }
          }

 e        return;
      }
```

Figure 2.7: Example program for demonstrating the goal-oriented approach

a starting point, the critical false branch from node 4 is taken. The search will fail, since small exploratory moves of `a` will have no effect on the objective function associated with this condition, which is concerned only with the value of the internal variable `b`. The landscape of the objective function in this region of the search space is flat (Figure 2.10).

In this example, one could attribute the failure to the use of a local search technique. A global search technique such as a genetic algorithm is likely to sample the input domain more thoroughly and find the required value of `a`. The local search could incorporate a series of restarts. However, it may be that the required path up to the target node is found with some very low probability. Even genetic algorithms will have trouble with these search spaces (see Section 2.3.5).

Korel realized that the problem might be solved by simply identifying a sequence of nodes which need to be executed prior to the test goal. In the example, if node 3 were to be executed before node 4, then the search would focus on the sloping part of the objective function surface. This concept is the basis of the *chaining approach*.

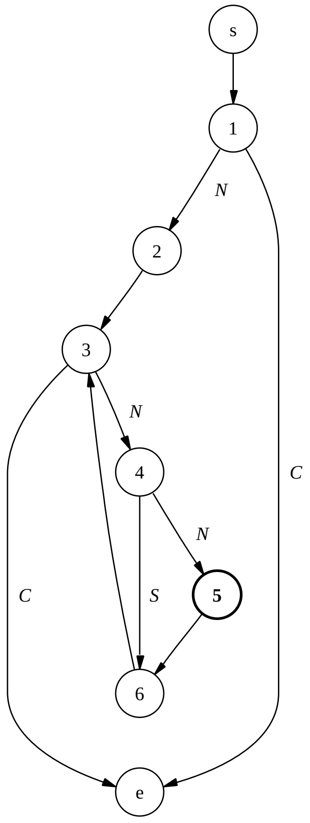


Figure 2.8: Control flow graph and branch classification for example program demonstrating the goal-oriented approach (Figure 2.7). Node 5 is the target. C represents a critical branch; S, a semi-critical branch; and N, a non-essential branch

```
CFG
Node
 s     void chaining_approach_example(int a)
       {
 1         int b = 0;

 2         if (a > 0)
           {
 3             b = a;
           }

 4         if (b >= 10)
           {
 5             // target
           }

           // ...
       }
```

Figure 2.9: Example program for demonstrating the chaining approach

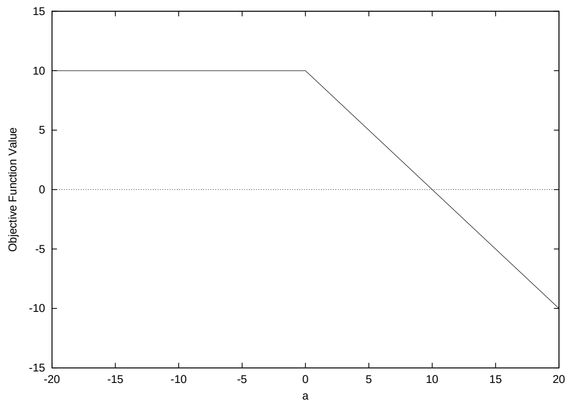


Figure 2.10: Objective function landscape for execution of node 4 as true for the example program for demonstrating the chaining approach of Figure 2.9

The Chaining Approach

The chaining approach [FK96a, Kor96, FK96b] uses the concept of an *event sequence* to find the sequence of program nodes that need to be executed prior to the target structure. The program nodes added to the event sequence are identified using data dependency analysis. This section briefly introduces the approach, which is central to this thesis, and discussed in finer detail in Chapter 4.

Recall from the last section that the search for inputs to execute the branching node 4 as true for the program of Figure 2.9 can fail when the value of `a` is negative, for example when `a = -10`. In this case the false branch from node 4 becomes critical. However, the local search is unable to find an input value of `a` so that the alternative true branch is taken, since exploratory moves from `-10` yield no change in values of the objective function associated with this branch. When inputs cannot be found to change the flow of control so that a critical branch (p, q) is avoided, p is declared as a "problem" node. Event sequences are generated by searching for "last definition" statements for variables used at the problem node. A last definition statement is simply a program node n that assigns a value to a variable which may be potentially used at the problem node p. For it to be a last definition therefore, a definition-clear path must exist between n and p. In the example, the variable used at node 4 is the internal variable `b`. This variable is defined at nodes 1 and 3. Therefore, two different event sequences are generated, one inserting an event where node 1 should be executed before node 4, avoiding node 3; and one where node 3 should be executed before node 4.

The latter event sequence (execution of node 3 before node 4) leads to success. Assume the input vector is still `(a = -10)`. Control flow is driven down the false branch at node 2. Now that the search aims to execute each node in the current event sequence in succession, this branch is now regarded as critical. The alternating variable method is invoked so that the true branch might be taken. Increments in `a` have a positive effect on the objective function associated with the true branch. Eventually the input `(a = 1)` is found. Flow of control is now driven down the critical false branch at node 4. However, exploratory moves of `a` now have an effect on the objective function associated with this branch. An increment of `a` leads to an improvement in the cost function, until eventually the vector `(a = 10)` can be found, and the goal node - node 5 - is executed.

It was found that the chaining approach could generate test data for a larger class of test goals in programs than the goal-oriented approach [Kor96, FK96a].

However the use of local search means that test data cannot always be found for complex structures that have complex search spaces.

2.3.4 Applying Simulated Annealing

The work of Tracey and co-authors [TCMM98, TCM98b] applies simulated annealing to structural test data generation, in the hope of overcoming some of the problems associated with the application of local search. In this work, test data can be generated for specific paths, or for specific statements or branches.

In order to apply simulated annealing, a neighbourhood structure has to be defined for the various different input variable types. For integer and real variables, the neighbourhood is simply a defined range of values around each individual value. Since the ordering of values is not significant for boolean and enumerated types, all values for these variables are considered as neighbours.

The objective function is simply the branch distance of the required branch when control flow diverges away from the intended path, or away from the target structure down a critical branch. The objective functions used (Table 2.2) are in principle identical to those employed by Korel, except the use of a non-zero positive failure constant K - which is always added if the branch predicate evaluates to false - removes the need to use a relation *rel* within the function. In this way, the objective function always returns a value above zero if the predicate is false, and zero when it is true.

In order to reduce the chances of the search becoming stuck in local optima, Tracey drops the constraint employed by Korel that the newly generated solution must conform to an already successful sub-path. However, the means of doing this results in the search losing some information about its progress. This is because solutions which diverge away from the target down earlier critical branches are assigned similar objective values to those diverging away at a later stage. This can be demonstrated with the example of Figure 2.11. For the target statement at node 3, the false branches from nodes 1 and 2 are critical. Under Korel's scheme, if the current solution is `(i=10, j=-1)`, diverging down the critical branch from node 2, the vector `(i=9, j=-1)` would not be given consideration, because the already successful sub-path up to node 2 is violated. This is due to the fact that this input vector takes the earlier critical branch at node 1. However in Tracey's method, a move can take place between solutions, and furthermore, the solutions are rewarded identical objective values - since the distance values taken at the different branching nodes are the same.

Table 2.2: Tracey's objective functions for relational predicates. The value K, $K > 0$, refers to a constant which is always added if the term is not true

Relational Predicate	Objective Function *obj*
Boolean	if $TRUE$ then 0 else K
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$\neg a$	Negation is moved inwards and propagated over a

```
CFG
Node
 s    void landscape_example(int i, int j)
      {
 1        if (i >= 10 && i <= 20)
          {
 2            if (j >= 0 && j <= 10)
              {
 3                // target statement
                  // ...
              }
          }
      }
```

Figure 2.11: Example program for comparing different objective functions

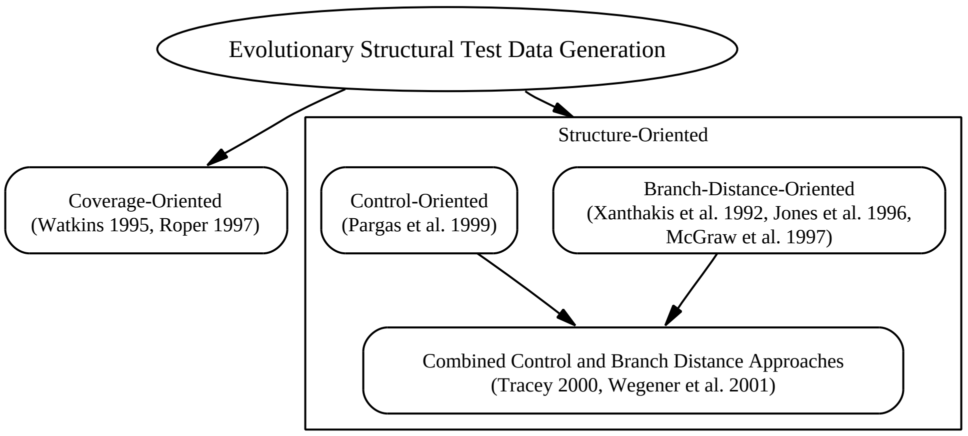

Evolutionary Structural Test Data Generation
Structure-Oriented
Coverage-Oriented
(Watkins 1995, Roper 1997)
Control-Oriented
(Pargas et al. 1999)
Branch-Distance-Oriented
(Xanthakis et al. 1992, Jones et al. 1996,
McGraw et al. 1997)
Combined Control and Branch Distance Approaches
(Tracey 2000, Wegener et al. 2001)

Figure 2.12: Classification of dynamic structural test data generation techniques using evolutionary algorithms

2.3.5 Applying Evolutionary Algorithms

The first work applying evolutionary algorithms to generate structural test data is that of Xanthakis *et al.* [XES+92]. Up until this point, work on structural test data generation had largely focused on finding input data for specific paths or individual structures with programs, such as branches or statements. Initially, however, techniques using genetic algorithms took slightly different directions.

A Classification of Techniques

Different techniques applying evolutionary algorithms to structural test data generation can be categorized on the basis of objective function construction (Figure 2.12).

Coverage-oriented approaches reward individuals on the basis of covered program structures. In the work of Roper [Rop97], an individual is rewarded on the basis of the number of structures executed in accordance with the coverage criterion. Under this scheme, however, the search tends to reward individuals that execute the longest paths through the test object. Guidance is not given for structures that are unlikely to be covered by chance, for example deeply nested structures, or branch predicates that are only true when an input variable has to be a specific value from a large domain.

The work of Watkins [Wat95] attempts to obtain full path coverage for programs. The objective function penalizes individuals that follow already covered paths, by assigning a value that is the inverse of the number of times the path has already been executed during the search. The direction of the search, therefore, is under constant adaptation. However, the penalisation of covered paths, in itself, provides little guidance to the discovery of new, previously unfound paths. The results show that in comparison with random testing, the genetic algorithm approach required an order of magnitude fewer tests to achieve path coverage for two experimental programs. However, both of these programs are of a simple nature, containing no loops. Furthermore, the input domains were artificially restricted for the search.

In general, the problem with coverage-oriented approaches is the lack of guidance provided for structures which are only executed with values from a small portion of the overall input domain. Therefore, it is difficult to expect full coverage with these techniques for any non-trivial program.

Structure-oriented approaches follow similar lines to the earlier work of Korel, and take a 'divide and conquer' approach to obtaining full coverage. A separate search is undertaken for each uncovered structure required by the coverage criterion. Structure-oriented techniques differ in the type of information used by the objective function. These can be categorized as either *branch-distance-oriented*, *control-oriented*, or *combined* approaches.

Branch-distance-oriented approaches exploit information from branch predicates, in a similar style to earlier work by Miller and Spooner, and later Korel. In the work of Xanthakis *et al.* [XES$^+$92], genetic algorithms are employed to generate test data for structures not covered by random search. A path is chosen, and the relevant branch predicates are extracted from the program. The genetic algorithm is then used to find input data that satisfies all the branch predicates at once, with the objective function summing branch distance values. However, this scheme suffers from similar problems suffered by the work of Miller and Spooner. Furthermore, the need to select a path is a burden on the tester. In the work of Jones *et al.* [JSE96] for obtaining branch coverage, a path does not need to be selected. The objective function is simply formed from the branch distance of the required branch. However, no guidance is provided so that the branch is actually reached within the program structure in the first place. McGraw *et al.* [MMS01] alleviate this problem for condition coverage, by delaying an attempt to satisfy a condition within a branching expression until previous individuals have been already found which reach the branching node in question. The initial generation for the target condition is then seeded with these individuals. This scheme, however, is inefficient if test data is required for the coverage of one, specific condition.

The earlier work of Korel had already removed the need for the tester to select a path. Since new test data considered by the search had to conform to the successful sub-path already found, explicit control-oriented information regarding the target did not need to be included in the objective function. However, such rigid constraints increase the chances of the search becoming stuck in local optima, and it would be better if more feedback could be provided via the objective function. This is the problem addressed by *control-oriented* approaches.

With *Control-oriented approaches*, the objective function considers the branching nodes that need to be executed in some desired way in order to bring about execution of the desired structure. The approach of Jones *et al.* [JSE96] to loop testing falls into this category. Here, the objective function is simply the difference between the actual and desired number of iterations. In the work of Pargas *et al.* [PHP99], for statement and branch coverage, the control dependence graph of the test object is used. The sequence of control dependent nodes is identified for each structure. These are the branching nodes that must be executed with a specific outcome in order for the structure to be reached. The objective value of an individual is simply assigned as the number of control dependent nodes executed as intended. Recall that the branch leading away from the target at a control dependent node is identified as a *critical branch* in Korel's work. The measure used by Pargas *et al.* is therefore equivalent to the number of critical branches successfully avoided by the individual.

The problem with using control information only for the purposes of the objective function are the plateaux that form on the objective function landscape. The objective function gives no guidance as to how to change the flow of execution at control dependent nodes, since no distance information is exploited from branch predicates. Take the simple example of Figure 2.11. The target is node 3, which is control dependent on node 2, which in turn is control dependent on node 1. Let *dependent* be the number of control dependent nodes for the current target, and *executed* the number of control dependent nodes successfully executed in the required manner. A minimising version of the objective function of Pargas *et al.*, can be computed as ($dependent - executed$). However, in this scheme, *every* individual diverging away from the target at node 1 receives an objective value of 2, with *every* individual diverging at node 2 receiving a value of 1. The landscape for the minimising version of the objective function for the example is seen in Figure 2.13. This landscape has three plateaux. For individuals not satisfying one or more of the branch predicates, no guidance is given as to how to descend down the landscape to solutions that are closer to executing the target. Along these horizontal planes, the search becomes random.

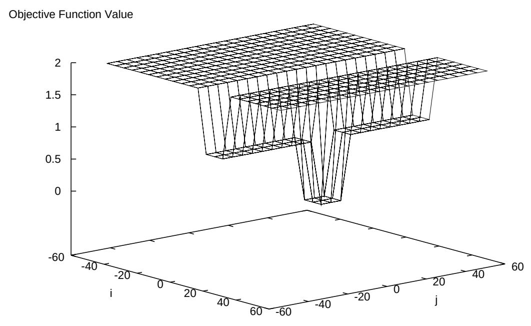


Figure 2.13: The objective function landscape of Pargas *et al.* [PHP99] for example of Figure 2.11

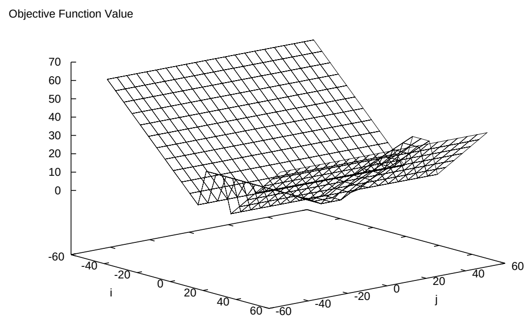


Figure 2.14: Objective function landscape of Tracey [Tra00] for example of Figure 2.11

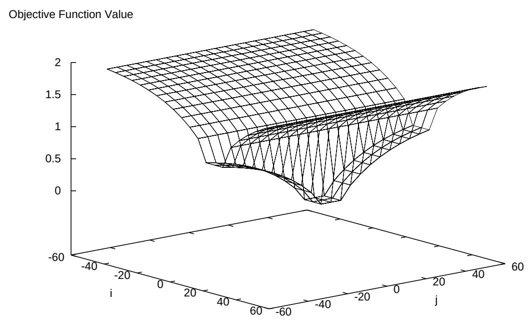


Figure 2.15: Objective function landscape of Wegener *et al.* [WBS01] for example of Figure 2.11

Combined approaches make use of both branch distance and control information for the objective function. The work of Tracey [Tra00] builds on previous work which used simulated annealing. The strategy for combining both techniques is as follows. The control dependent nodes for the target structure are identified. If an individual takes a critical branch from one of these nodes, a distance calculation is performed using the branch predicate of the required, alternative branch. This is computed using the functions of Table 2.2 (and Table 2.3 for *and* and *or* logical connectives). Tracey then uses the number of successfully executed control dependent nodes to scale branch distance values. Let *branch_dist* be the branch distance calculation performed at the branching node where a critical branch was taken. The formula used by Tracey for computing the objective function is:

$$\left(\frac{executed}{dependent}\right) \times branch\_dist$$

Unfortunately, this scheme can lead to unnecessary local optima in the objective function landscape. For the example of Figure 2.11, this is evident by the valleys in the objective function landscape along $i = 9$ and $i = 21$ where $-3 \leq j$ and $j \geq 13$, as seen in Figure 2.14.

The objective function of Wegener *et al.* [WBS01, WBP02] normalizes branch distance values *branch_dist* into the range 0-1 using the following function [Bar02]:

$$normalize\_bd(branch\_dist) = 1 - 1.001^{-branch\_dist} \tag{2.1}$$

This is combined with another value called the *approximation level*, referred to throughout this thesis as the *approach level*, calculated as follows:

$$approach\_level = dependent - executed - 1 \tag{2.2}$$

The minimising objective function is zero if the target structure is executed, otherwise, the objective value is computed as:

$$approach\_level + normalize\_bd(branch\_dist) \tag{2.3}$$

The resulting objective function landscape has a similar form to that of Pargas *et al.* (Figure 2.15). However, the extra information provided by the branch distance calculation prevents the formation of plateaux at each approach level. For this example, the result is a sweeping landscape from each level to the next level downwards.

Objective Functions for Different Structural Coverage Criteria

The work detailed so far for structural test data generation has mainly addressed statement, branch or condition coverage. In the work of Wegener *et al.* [WBS01], several new structure-oriented objective functions were introduced for previously unexplored coverage types. For this purpose, structural criteria are divided into four categories:

- node-oriented
- path-oriented
- node-path-oriented
- node-node-oriented

The basic form of the (minimising) objective function is:

$$approach\_level + normalize\_bd(branch\_dist)$$

The strategy in which *approach_level* and *branch_dist* are computed varies according to the coverage type in question.

Node-oriented criteria aim to cover specific nodes of the control flow graph, for example statement coverage. The strategy for node-oriented methods was discussed in the last section. The approach level is calculated on the basis of the number of control dependent nodes for the target lying between nodes covered by the individual and the target node itself. At the point where control flow diverges down a critical branch, the branch distance is calculated using the predicate of the alternative branch.

Path-oriented criteria require the execution of specific paths through the control flow graph. There are two possible ways to calculate the objective function. One method is to calculate the approach level on the basis of the length of identical initial path section, with the branch distance calculation performed using the predicate at the first diverging branch. An alternative strategy considers all identical path sections for the approach level, with the branch distance calculation an accumulation of distance calculations made at each point of divergence from the intended path. Wegener *et al.* report superior results with the latter method [WBS01].

Node-path-oriented criteria include branch coverage and LCSAJ (linear code sequence and jump) coverage, where a node and a specific subsequent path must be executed. The objective function is a combined node-oriented and path-oriented calculation. Calculations for individuals not reaching the initial node are treated as for node-oriented criteria. For individuals reaching the initial node, a path-oriented calculation is additionally applied.

Node-node-oriented criteria aim to execute a certain sequence of nodes through the control flow graph, without the specification of a concrete path between each node. This includes data-flow-oriented coverage types such as *all-defs* and *all-uses* criteria. In this case, the objective function is a cumulative node-oriented strategy. Calculations for individuals failing to reach the first node are carried out as for node-oriented methods, with individuals reaching the subsequent node having additional calculations carried out at these further nodes.

Control-Related Problems for Objective Functions

The provision of guidance to structures nested within loops presents a problem which can be demonstrated with Figure 2.16. The target is the execution of node 3. Node 3 is control dependent on node 2, but the false branch from node 2 can also lead to the execution of node 3 via iteration of the loop. The calculation of the branch distance at node 2 in the first iteration of the loop, therefore, may not provide a good objective measure. In the example, if the input variable `i` is `1`, the objective value is taken in the first iteration, when `n` is `0`. However, the individual is closest to executing the target statement in the last iteration of the loop, when `n` is `10`. Furthermore, when the input value of `i` is `0`, the individual will be deemed to have missed the target, when the target is actually executed in the last iteration of the loop. In order to circumvent this problem, Tracey [Tra00] examines the branch distance during each iteration of the loop and uses the minimum branch distance obtained for the purposes of computing the final objective value.

A further problem is the assignment of approach levels for some classes of program with unstructured control flow. Baresel *et al.* [BSS02] present the example of Figure 2.17. The target of the search is node 6. However, there are three different control dependent paths through to node 6 from node 1 (Figure 2.18), and two control dependent paths from node 2 (control dependency was defined on page 22). Consequently there are two approach level possibilities for node 1 (since two of the three paths are of the same length), and two possibilities for node 2. Two plausible solutions to this problem include *optimistic* and *pessimistic* approach level allocation strategies. In an optimistic strategy, a control dependent branching node is allocated its approach level on the basis of the shortest control dependent path from itself to the target node. In this way node 4 is assigned an approach level of 0 on the basis of the direct path through

```
CFG
Node
 s    void loop_example(int i)
      {
          int n;
 1        for (n=0; n <= 10; n++)
          {
 2            if (n == 10 && i == 0)
              {
 3                // target statement
              }
          }
 e    }
```

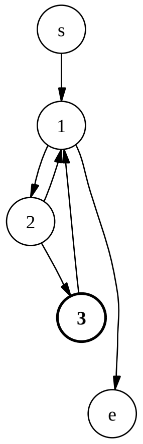


Figure 2.16: Program with a loop and its control flow graph

to 6, thereby receiving the same level as node 5. In a pessimistic strategy, a branching node is allocated its approach level on the basis of the longest control dependent path to the target node. In this scheme node 4 would be assigned an approach level of 1 on the basis of the path through node 5. Both optimistic and pessimistic schemes were put to the test in initial experiments by Baresel *et al.* [BSS02]. Whilst they show that the different schemes have different effects on the progress of the search, they were unable to conclude from the experiments which strategy works best in general. Thus, this problem is still open to question.

Branch-Distance-Related Problems for Objective Functions

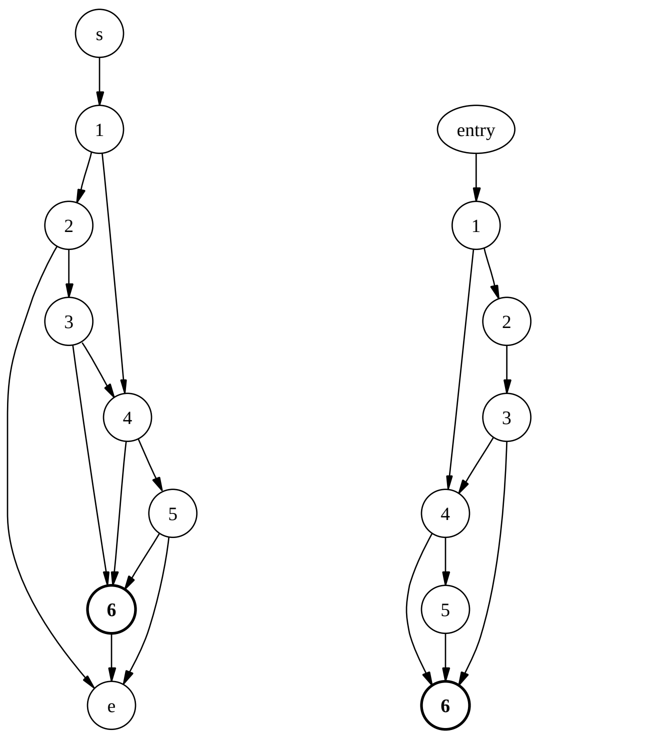
Although global search techniques are more robust than local searches in objective function landscapes containing local optima and plateaux, they will still struggle in hostile search landscapes containing large plateaux or several local optima.

In particular, plateaux can be induced on the search space through the use of internal “flag” variables in branch predicates. A flag is simply a boolean variable. When flag variables are involved in branch predicates, the resulting objective function landscape consists of two plateaux - one for the true value and one for the false value. In such situations, the evolutionary search performs no better than a random search.

Figure 2.19 demonstrates this with an example. For the true branch to be executed, the flag must be true. However, the objective function gives no guidance to how the true value is brought about. The plateau induced on the

```
CFG
Node
 s    void unstructured_example()
      {
          switch(a)
          {
 1            case 1:
 2                if (cond_1)
                      return;
 3                if (cond_2)
                      break;
 4            case 2:
 5                if (cond_3)
                      break;
                  return;
          }
 6        // target statement
 e    }
```

Figure 2.17: Program with unstructured control flow



a) Control flow graph b) Control dependence graph

Figure 2.18: Control graphs for program with unstructured control flow of Figure 2.17

```
flag = (d == 0);

if (flag)
    result = 0;
else
    result = n / d;
```

Figure 2.19: Example program with a flag variable

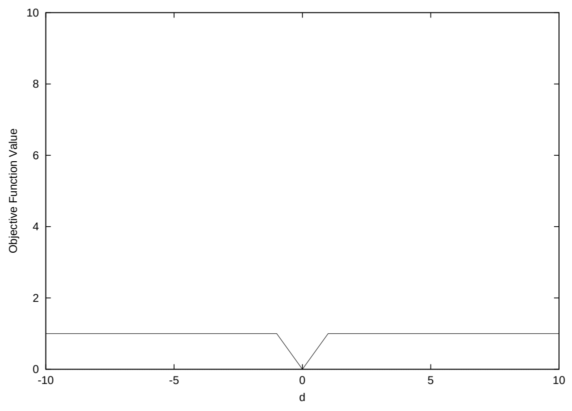


Figure 2.20: Objective function landscape for the example program with flag variable of Figure 2.19

objective function landscape can be seen in Figure 2.20.

Bottaci [Bot02] proposes a solution for a special case of flag problems similar in form to the example of Figure 2.19, where the value of the flag is determined by a predicate. In this work it is suggested that the predicate used for the distance calculation is substituted by the predicate used in assigning the flag value. Essentially the objective function landscape becomes that of Figure 2.21, which provides more guidance to the required test data. However, flags are more commonly assigned constant true or false values, as seen in Figure 2.22. In this case the expression leading to the true assignment is used to control the assignment. [Note that the true branch from node 4 would have already been executed if test data had already been found to execute the preceding true branch from node 2. However, for simplicity, this possibility is ignored for the purposes of this example, and others in this section].

Harman *et al.* [HHH$^+$02, HHH$^+$04] suggest the use of a program transformation to remove internal flag variables from branch predicates, replacing them with the expression that led to their determination. In the transformed version of the program, the branch predicate is flag-free, and consequently plateaux

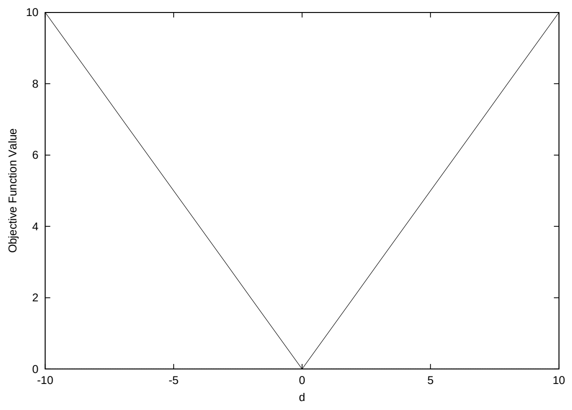


Figure 2.21: Objective function landscape for the predicate `d == 0` for the example program with flag variable of Figure 2.19

CFG Node	
1	`flag = false;`
2	`if (d == 0)`
3	`    flag = true;`
4	`if (flag)`
5	`    result = 0;`
	`else`
6	`    result = n / d;`

Figure 2.22: Alternative version of the flag example program

```
flag = false;
if (d == 0)
    flag = true;

if (d == 0)
    result = 0;
else
    result = n / d;
```

Figure 2.23: Flag removed from branch predicates of alternative version of flag program (Figure 2.22) via program transformation

induced by the flag are also removed. Figure 2.23 shows a possible transformation of the program of Figure 2.22. Note that although the flag is removed from the branch predicate, it otherwise remains present in the program, in case it has a future purpose in a later statement. The objective function at the new branch predicate now has the more useful landscape of that of Figure 2.21. The transformed program is merely a means to an end, and can be discarded once the required test data has been found.

Baresel *et al.* [BBHK04] extend the transformation approach for internal flags assigned within loop structures. Two approaches are presented - a "coarse-grained" transformation and a "fine-grained" transformation. Both forms of transformation replace the original condition using the flag variable with a predicate of the form `counter == fitness`, where `counter` is a variable incremented on each iteration of the loop, and `fitness` is a variable which is incremented if a loop iteration was evaluated in a "successful" manner. Whether an iteration can be counted as successful depends on the path taken through the loop iteration, and whether this path supports the final value of the flag required for executing the evaluating the original condition in the desired way. For example, an iteration which assigns a false value to a flag required as true would not result in an increment of the `fitness` variable; whereas the avoidance of the assignment would. In this way, the search receives a higher level of guidance to the input values which evaluate the original condition using the flag in the desired manner. This is because the objective function landscape now corresponds to the predicate `counter == fitness` rather than the landscape containing the flag, which contains plateaux. The difference between the coarse-grained transformation and the fine-grained transformation lies in the increment of the `fitness` variable within the loop. The coarse-grained transformation simply increments the counter in a uniform fashion. The fine-grained approach uses distances of key branch predicates used within the loop to assign

```
if (d == 0)
    r = 0;
else
    r = 1 / d;

if (r == 0)
    // target branch
```

Figure 2.24: Deceptive objective function example program (adapted from Reference [Har02])

flag values. The latter approach provides a greater level of guidance to the search, enabling it to find test data in cases where the coarse-grained approach could not, or faster in cases where both approaches were able to find test data.

An alternative approach to handling internal flag variables is that of Baresel *et al.* [BS03]. The basic idea of this method is to use data flow analysis to statically decide which assignments to the flag need to be executed and which assignments need to be avoided so that the flag will have the correct value in order for the target structure to be covered. An objective function is then derived from the branch distances leading to the desired assignments and branch distances which avoid undesired assignments. For the example of Figure 2.22 where the true branch from node 4 is required to be executed, is it clear that node 3 needs to be executed before node 4 is reached. The derived objective function uses similar principles to the *node-node* oriented functions, discussed in Section 2.3.5, using the true branch distance of node 2 to first guide the search to the true assignment to the flag, and then onto node 4. It is stated that the approach has problems avoiding unrequired assignments to flags within loop bodies [BS03].

Aside from problems of local optima and plateaux appearing in the objective function landscape, it is entirely possible for the branch distance calculation to deceive the search. Consider the example of Figure 2.24. The goal is to execute the true branch of the final branching node, whose branch predicate is `r == 0`. However, unless `d` is zero, `r` will not be zero. The objective function works to guide the search away from `d` being equal to zero, since increasing values of `d` decrease values of `r` deceiving the search into believing it is getting closer and closer to zero, as depicted by the objective function landscape (Figure 2.25).

A further problem can occur with nested branch predicates as seen with the example of Figure 2.6. In this example, input data must be found satisfying `a == b` before the solution to `b == c` and `c < 0` can be attempted. Once input

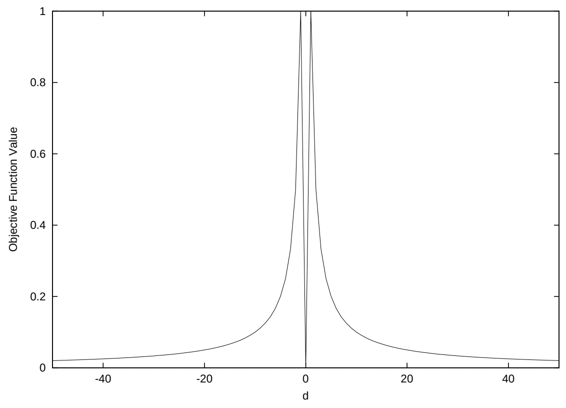


Figure 2.25: Landscape for the deceptive objective function example program of Figure 2.24

data is found for one or more of the conditions, the chances of finding input data that also fits subsequent conditions decreases. This is because a solution for subsequent conditions must be found without violating any of the earlier conditions. This leads to poor search performance. Ideally, all of the conditions should be evaluated at once. This was the solution employed by Baresel *et al.* [BSS02], leading to much improved search performance. Here, none of the values `b`, `c` or `d` are modified between the branching statements, and so all predicates could be evaluated at the first branching statement. Such a situation could be established through the use of data dependency analysis [BSS02].

A similar problem occurs with the use of short circuit evaluation of atomic conditions with branch predicates using operators such as `&&` and `||` in C. In such situations the evaluation of the overall predicate breaks off early if the end result has already been determined. Therefore, during the process of searching for test data, the individual conditions have to be attempted one after the other. For example:

```
if (a == b && b == c && c < 0)
{
    // ...
}
```

Again, it would be preferable to evaluate all of the conditions at once, as performed by Baresel *et al.* [BSS02]. In this situation, care needs to be taken when side effects appear in any of the conditions. A solution here might be to apply a side-effect removal program transformation [HHZM01, HHZ$^+$02] first. Alternatively, variable values could be saved into temporary variables inserted immediately before the branching statement, and restored after the side-effect if the condition would not normally have been evaluated.

Applying Variable Dependence Analysis

Harman *et al.* [HFH$^+$02] apply variable dependence analysis to determine the subset of input variables that cannot affect the outcome at a branch predicate. In this way, the search space can be reduced, increasing the chances of finding a solution - and potentially finding it faster. Take the triangle example of Figure 2.4 once more. For branching node 1, only the input variables `a` and `b` are relevant. Variable `c` cannot affect the outcome at this node, and as such does not need to be included in the search. For branching node 5, all input variables are relevant, because `b` may have determined the outcome of `a` during the prior nodes 1-4. These ideas are similar to Korel's influences graph [Kor90] (see Section 2.3.3), except the information is statically computed for each structural target. The variable dependence analysis information can also be used to compute a slice of the program with respect to the structural target. A program slice [Wei84] is a smaller version of the original program which only contains the statements of interest according to some slicing criterion. In this case the criterion involves the removal of all statements that cannot affect the attainment of the desired structure. Such slices are potentially useful since they can cut down the time required to execute the program and evaluate individuals of the search.

Use of Evolutionary Algorithms: Encodings and Operators

Early work in applying genetic algorithms to structural test data generation used binary encodings. Jones *et al.* [JSE96] found improvement in the use of a gray code.

However, it is common that variables will often only have valid values within a subset of the possible bit patterns at the binary level. In addition to the range imposed on an ordinal type by a compiler, input variables are often restricted to a certain range by the context of its application. One problem that can occur with binary encodings is the corruption that can occur with restricted types through the actions of the crossover and mutation operators. This problem was raised by Tracey [Tra00]. The following shows two chromosomes (26, 81) and (56, 43) representing two integer variables restricted between 1 and 100. Crossover at locus 8 yields two offspring - (26, 107) and (56, 17).

$$\begin{array}{l|l} \texttt{00110101} & \texttt{010001} \\ \texttt{01110000} & \texttt{101011} \end{array} \rightarrowtail \begin{array}{l} \texttt{00110101101011} \\ \texttt{01110000010001} \end{array}$$

The final variable of the former chromosome is now out of range. One solution might be to restrict the crossover points to the boundaries of each variable, making it impossible for a variable value to go out of range. However the chromosome can still be damaged by the mutation operator. A possible solution is to repair or penalize invalid individuals. An alternative is to use a real-valued encoding. This is the decision taken by Tracey [Tra00] and Wegener *et al.* [WBS01]. For real-valued encodings, crossover is naturally restricted to the boundaries of each variable. For example:

$$\begin{array}{r|l} \texttt{26} & \texttt{81} \\ \texttt{56} & \texttt{43} \end{array} \quad \rightarrowtail \quad \begin{array}{ll} \texttt{26} & \texttt{43} \\ \texttt{56} & \texttt{81} \end{array}$$

The mutation operator can also be based on number creep (introduced in Section 2.2.3), taking care to ensure that each value is not shifted out of its required range. The use of a real-valued encoding also removes the need to encode and decode the input vector into and out of a binary format.

Finally, the test generation system of Wegener et al. employs competition and migration amongst subpopulations [WBS01, WBP02].

2.4 Functional (Black-Box) Testing

This section discusses the application of metaheuristic search techniques to the testing of the logical behaviour of a system, as described by some form of specification.

2.4.1 Generating Test Data from a Z Specification

Jones *et al.* [JSYE95] generate test data for the triangle classification program, using a Z specification [Spi92]. The state space of the system is described in a schema named *Triangle0*, which declares three input integer variables to represent the three sides of the triangle ($x?$, $y?$ and $z?$). This schema also describes invariants over the inputs to check that the lengths are within a specified range, and that the side lengths represent a valid triangle. These checks are also included in two other operations declared as $NumError$ and $TriangleError$. Four further operations decide if the triangle is scalene ($ScalTri$), equilateral ($EquiTri$), isosceles ($IsosTri$) or right-angled ($RightTri$).

Using these schema, the whole system can be declared as:

$$\begin{aligned} Triangle ::= \; & (Triangle0 \wedge EquiTri) \vee (Triangle0 \wedge IsosTri) \vee \\ & (Triangle0 \wedge ScalTri) \vee (Triangle0 \wedge RightTri) \vee \\ & NumError \vee TriangleError \end{aligned}$$

For the purposes of test data generation, each disjunct is considered as a *route* through the system. Genetic algorithms are used to search for test data that satisfies each route.

The fitness function rewards individuals that come close to satisfying the conjuncts in each route. In the case of an equilateral triangle, the predicates to be satisfied include invariants from the state space schema conjuncted with those of the $EquiTri$ schema $((x? = y?) \wedge (y? = z?))$. Each conjunct is evaluated using a distance based approach, in a similar fashion to the branch distance calculations used in Structural Testing. The overall fitness of the route is the summation of the distances for each of its conjuncts.

The results report successful test data generation by the genetic algorithm for each of the routes under examination, namely $ScalTri$, $EquiTri$, $IsosTri$ and $RightTri$. However the example is small and not general enough to establish its usefulness. Furthermore, only a small subset of Z is used, and this is limited to the use of relational operators only.

2.4.2 Testing Specification Conformance

The last section showed how test data could be generated from a formal specification. The work of Tracey *et al.* [TCM98a, Tra00] extends this idea. In their technique the conformance of the implementation to its specification is checked by executing the test object with the generated test data, and then validating the output against the specification.

The specification of the implementation is represented as a pre-condition, which defines valid inputs, and a post-condition, which defines the output. A failure is found when an input situation is discovered that satisfies the pre-condition of the function, but for which the outputs violate the post-condition. An objective function is derived which describes the “closeness” of the test data to uncovering such a situation, and metaheuristic search techniques are then employed to seek failures in the implementation.

As a simple example, take the wrapping counter function of Figure 2.26. This function implements a counter, which takes an integer value between 0 and 10, and returns the increment. If the input is 10, the counter wraps round to 0. The pre-condition for this function is simply:

$$n \geq 0 \wedge n \leq 10$$

The post-condition is:

$$(n < 10 \rightarrow r = n + 1) \vee (n = 10 \rightarrow r = 0)$$

```
int wrapping_counter(int n)
{
    int r;
    if (n >= 10)
        r = 0;
    else
        r = n + 1;

    return r;
}
```

Figure 2.26: Wrapping counter example program

Table 2.3: Tracey's objective functions for logical connectives, where $obj(c)$ is the individual cost of connective c

Connective	Objective Function obj
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
$a \Rightarrow b$	$obj(\neg a \vee b)$
	$\equiv min(obj(\neg a), obj(b))$
$a \Leftrightarrow b$	$obj((a \Rightarrow b) \wedge (b \Rightarrow a))$
	$\equiv obj((a \wedge b) \vee (\neg a \wedge \neg b))$
	$\equiv min((obj(a) + obj(b)), (obj(\neg a) + obj(\neg b)))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
	$\equiv min((obj(a) + obj(\neg b)), (obj(\neg a) + obj(b)))$

where n is the input value and r is the return value.

A constraint system is then derived to describe conditions of implementation non-conformance by taking the pre-condition in conjunction with the negated post-condition:

$$n \geq 0 \wedge n \leq 10 \wedge \neg((n < 10 \rightarrow r = n + 1) \vee (n = 10 \rightarrow r = 0)) \tag{2.4}$$

An objective function is derived to indicate how "close" failure is . This is constructed from the above constraint system using the rules in Tables 2.2 and 2.3:

$$\begin{gathered} obj(n \geq 0) + obj(n \leq 10) + \\ min((obj(n < 10) + obj(r \neq n + 1)), (obj(n = 10) + obj(r \neq 0))) \end{gathered} \tag{2.5}$$

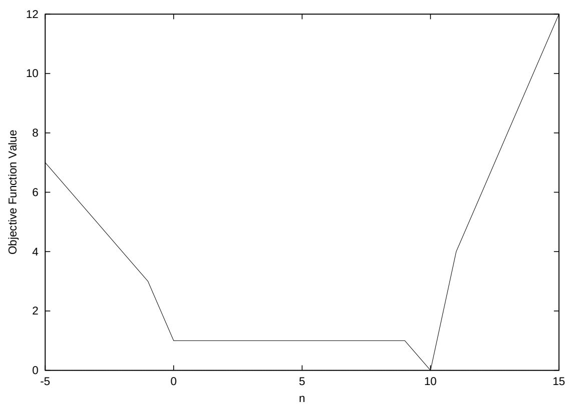


Figure 2.27: Objective function landscape for wrapping counter example of Figure 2.26, where $K = 1$

It was found that the landscapes of the objective functions derived from such constraint systems contained areas of plateaux. Figure 2.27 shows the objective function landscape for a faulty version of the program where the branch predicate `n >= 10` is replaced by `n > 10`. The objective function is zero when $n = 10$, indicating a fault. However, a plateau forms for values of n between 0 and 9. This results from the use of the *min* operator in the objective function. For $n < 10$, the objective value of the first operand, $obj(n < 10)+obj(r \neq n+1)$, is always K, which is always smaller than the objective value of the second operand $obj(n = 10) + obj(r \neq 0)$. It was found that guidance to the search could be improved by converting the constraint system to disjunctive normal form, and then using each disjunct as the basis of a separate search.

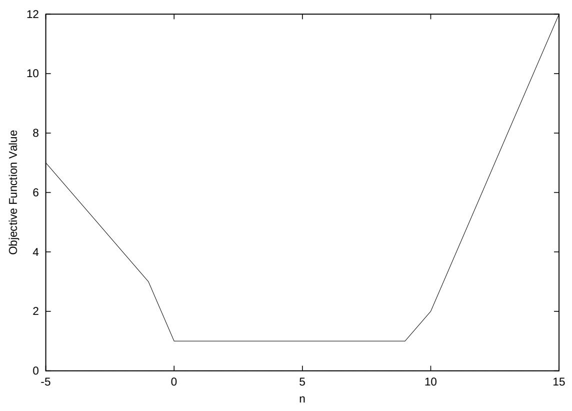
Conversion of the original constraint system (Equation 2.4) to disjunctive normal form gives two disjuncts:

Disjunct 1: $n \geq 0 \wedge n \leq 10 \wedge n < 10 \wedge r \neq n + 1$
Disjunct 2: $n \geq 0 \wedge n \leq 10 \wedge n = 10 \wedge r \neq 0$

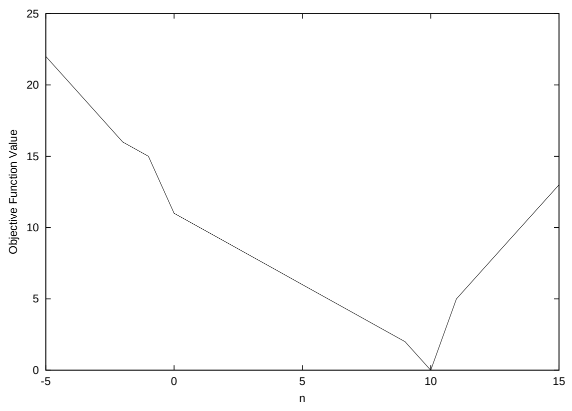
The objective functions for each disjunct, are, respectively:

Disjunct 1: $obj(n \geq 0) + obj(n \leq 10) + obj(n < 10) + obj(r \neq n + 1)$
Disjunct 2: $obj(n \geq 0) + obj(n \leq 10) + obj(n = 10) + obj(r \neq 0)$

Figure 2.28 shows the landscape for the faulty branch predicate `n >= 10` for the objective functions of disjuncts 1 and 2 respectively. As can be seen, the landscape for the second disjunct in the range $0 \leq n < 10$ gives more guidance



(a) Disjunct 1



(b) Disjunct 2

Figure 2.28: Objective function landscapes for individual disjuncts of the wrapping counter example of Figure 2.26, where $K = 1$

to the failure point when the objective value is zero.

Tracey [Tra00] applied this technique to the testing of a safety-critical nuclear primary protection system, written in Pascal. Two sub-systems were available for this evaluation. The first consisted of 36 pages of formal VDM-SL specification and the second 54 pages, with approximately 2000 lines of executable code. The pre- and post-conditions for each function of each sub-system were manually derived from the specification, with 733 different disjuncts obtained. A mutation testing tool was then used to generate mutant implementations of the code. Simulated annealing and genetic algorithms were then used as metaheuristic searches for the technique. Both searches killed 100% of approximately 170 non-equivalent mutants, outperforming hill climbing and random searches, which still achieved overall scores of over 90%.

Buehler and Wegener [BW03] use evolutionary algorithms to test specification conformance of an early version of an automated vehicle parking system. This system aims to automate parking of a vehicle lengthways into a parking space, using information from environmental sensors, which register surrounding objects. The individuals of the search are simply parking scenarios which describe the dimensions of a parking space, including collision areas, and the starting position of the car. The parking control unit is called with this data, and a parking manoeuvre is simulated. With a successful test being one which causes a collision, the objective function is simply the value of the smallest distance between the car and the collision area recorded during the simulation. In the experiment undertaken, approximately 900 scenarios were simulated, with more than 25 scenarios found leading to collisions. After analysis of these scenarios, it was discovered that the controller had difficulties with scenarios where the parking space was some distance away and the starting position was already near to the collision area on one side. A fault was also detected with the simulation environment, where it was found that calculations involving the position of the car were too imprecise. This led to further simulated impacts with the collision area.

Baresel *et al.* [BPS03] test Simulink and Stateflow models which require input signal sequences to be generated. One problem in this domain is the generation of realistic signals and their potential length, which could result in a very large search space. Baresel *et al.* propose a novel solution by building the overall signal from a series of simple signal types, for example sine, spline and linear curves. The search space then becomes the set of parameters used to construct a signal section built from a base signal, for example its amplitude and length. This guarantees the generation of realistic input signals, as well as keeping the size of the search space relatively compact. The Distronic cruise control system was tested using this technique. This system senses the approach to slower vehicles and automatically slows the car down to maintain a safe following distance. The objective function checks for violations of the requirements, by checking dependencies between output signals, checking for output signal boundary violations and checking signal maximal overshoot and settlement time. For Distronic, tests revealed that the system broke a maximal speed violation under certain input conditions.

2.5 Grey-Box Testing

Grey-box testing combines both structural and functional information for the purposes of testing.

2.5.1 Assertion Testing

The work of Korel and Al-Yami [KAY96] attempts to find test cases that violate assertion conditions, which can be embedded by the programmer into the program code. Assertions specify constraints that apply to some state of a computation. When an assertion evaluates to false, an error has been found in the program. Assertions can be embedded within comment regions, either as boolean conditions, for example:

```
/*@ i > 0 and i <= 10 @*/  // assertion
i ++;                       // program statement
```

or, as executable code. When assertions are embedded as blocks of executable code, a special variable `assert` is used. This is assigned true or false values to denote the correct or incorrect state of the assertion. For example, the following assertion checks that the elements of an array are sorted in ascending order:

```
/*@
assert = true;
for (i = 0; i < len-1; i++)
{
    if (a[i] > a[i+1])
        assert = false;
}
@*/
// ... normal program code ...
```

Korel and Al-Yami showed how the search for test data to falsify an assertion reduced to the problem of executing a specific statement in the program. First, assertions are stripped out of the code. For boolean conditions, code is generated and placed in the assertion's original position. The assertion condition is then negated. This new condition is the condition which represents a violation, and therefore, the finding of a fault. This is then converted to disjunctive normal form. A series of nested `if` statements are then generated for each condition within each individual disjunct. If each `if` statement is evaluated as true, the violation is reported. For example, take the assertion condition $(a < b \wedge \neg(b = c \wedge c = d))$. The negated form of the assertion is $(a \geq b \vee (b = c \wedge c = d))$. The following code is generated for this negated condition (which is already in disjunctive normal form):

```
if (a >= b)
    report_violation();
if (b == c)
    if (c == d)
        report_violation();
```

The goal of the search is then to execute one of the `report_violation()` statements.

For assertions appearing as code, the assertion code is formed into a function, with the original assertion comment region replaced with a call to that function. The goal is then to execute a false assignment to the `assert` variable statement within the function, and thereafter avoiding all true assignments to the variable.

The process of test data generation is performed using the chaining approach (Section 2.3.3). In addition to programmer embedded assertions, Korel's tool automatically generates assertions for run-time errors such as division by zero errors, array boundary violations and overflow errors. The tool also tries to find input data to stimulate error conditions where variables are not initialized, yet used in some following program statement.

In initial experiments, nine original Pascal programs were embedded with assertions. Twenty-five faulty versions were then produced. With these experiments, it was found that inputs could be found to violate an assertion - and thereby reveal a fault - 92% of the time.

2.5.2 Exception Condition Testing

Tracey *et al.* [TCMM00, Tra00] built on the ideas of Korel and Al-Yami, using genetic algorithms and simulated annealing to generate input data to test the handling of run-time error conditions in code. In many languages, such as C++, Java and Ada, these run-time errors are known as *exceptions*. These languages provide explicit exception-handling constructs so that exception-related code can be separated from the main logic of the program. Tracey *et al.* generate test data for the raising of the exception, and then for the structural coverage of the exception handler. As with the work of Korel, both problems reduce to the problem of the execution of a certain statement (i.e. the statement which triggers the exception via a `throw` or `raise` statement), or a sequence of statements through the code (the raising of the exception followed by coverage of some structural element within the exception-handler). Experiments were undertaken with seven simple programs of no more than two hundred lines of code. It was found that metaheuristic techniques could generate test data to raise almost all the exception conditions contained within the code, and full branch coverage of exception handlers where they existed. An industrial experiment was also undertaken on an engine controller. Here, test data was generated which raised a variety of exception conditions. However it was found that these exceptions could not be raised in practice, since input situations had been generated which were not possible during actual operation of the system.

2.6 Non-Functional Testing

To date, search-based testing effort in the area of non-functional testing has concentrated on checking the best-case and worst-case execution times of real-time systems.

2.6.1 Execution Time Testing

The correct operation of a real-time system not only depends on its logical behaviour, but also its timing behaviour. In general, incorrect timing behaviour of a real-time system occurs when outputs are produced too early or too late. Execution time testing, therefore, involves attempting to find the worst-case execution time (WCET) or the best-case execution time (BCET) of a system in order to determine whether it is compliant with its timing constraints. This task is extremely difficult to achieve, since the timing behaviour of a piece of software is not only dependent on its internal structure but also the characteristics of the target hardware. At the software level, timing is dependent on the instructions used and their corresponding data items. The compiler can also introduce effects not apparent at source code level. At the hardware level, accounting for the actions of the target processor is extremely difficult when caching and pipelining operations need to be considered. As a consequence, the longest or shortest path through the program will not necessarily yield the longest or shortest execution time.

Static Analysis

Static analysis can be used to derive upper and lower bounds on WCET and BCET respectively, in order to try and ensure that timing schedules will be met. This is performed by examining the possible execution paths and then modelling timing behaviour at the hardware level. The primary step needs assistance from the programmer, since information is required regarding infeasible paths, and the maximum number of iterations for each loop appearing in the code. Unfortunately, the possibility of simulation errors and the need for human involvement make this an error-prone process [PN98, WPS99]. The result produced can also be extremely pessimistic in the case of WCET and optimistic in the case of BCET. Sometimes the estimates can vary from those observed in practice by a magnitude of ten times [Weg03].

Consequently, the calculations produced still need to be tested. Of course, tests derived to expose flaws in the logical behaviour are generally of little benefit in this domain.

Search-Based Execution Time Testing

Search-based execution time testing seeks input situations which invoke extreme execution times. The objective function is simply the execution time of the system as executed with some input. The search attempts to maximise the objective function in the case of WCET, and minimise it in the case of BCET. If a test case is found that violates the timing constraints, the search can be terminated.

Wegener *et al.* [WGG$^+$96] were the first to apply genetic algorithms to temporal testing. In their experiments [WSJE97, WG98] it is shown that genetic algorithms yield better results than random testing. A number of experiments with industrial test objects were carried out. A further experiment investigated six time-critical tasks in an engine control system [WPS00]. Genetic algorithms were again found to outperform random search, and also tests constructed by the developers themselves. The developer's tests never found the longest execution times, and in three cases the developer tests were worse than the random tests. Since the developers had internal knowledge of the system, these results were met with some surprise. Wegener *et al.* suggest this may be down to the use of system calls, linkage and compiler optimization whose effects on temporal behaviour could only be guessed with difficulty by the developers. Additional work by O'Sullivan *et al.* [OVW98] applies cluster analysis to determine when the search should be terminated. This technique decides if the search is converging on the basis of the distribution of individuals in the search space.

Puschner *et al.* [PN98] apply genetic algorithms to find WCET for seven programs with differing execution-time behaviour. The results are compared with those obtained by random search, upper WCET bounds found by static analysis, as well as "best effort" times, which were the researcher's own efforts to find input data to yield the WCET. The genetic algorithm was found to match or find longer times than the random search. The superiority of the genetic algorithm was particularly evident in large input domains. The genetic algorithm found similar times to the best effort analysis, in one case finding a longer time. Whilst upper bound times found by static analysis were never broken, they were matched on several occasions. In practice, this is unusual since the times provided by static analysis are generally too pessimistic or too optimistic for WCET and BCET respectively.

Tracey employs simulated annealing and genetic algorithms for finding the WCET of a handful of small, well-understood programs written in Ada, with known WCET behaviour [TCM98b, Tra00]. Each experiment was deemed to be a success if the technique executed the path through the program which yielded the already known WCET. It was found that genetic algorithms were more successful than simulated annealing, both of which outperformed hill climbing and random search. Overall, the genetic algorithm achieved success in fewer trials than simulated annealing. In this particular study, it was found that varying the parameters of the optimization techniques had little effect on the end result, apart from when the initial temperature was set too low for simulated annealing, where dependency on the starting position could not be lost.

Unfortunately, if a branch in the program is executed only with a low probability, the chance of a search technique executing it is low. If this branch is involved in a path leading to an extreme execution time, then the extreme execution time will not be found. Gross [Gro01] identifies a number of properties of programs which lead to low probability branches, for example high levels of nesting, branches that are only executed if an input variable is a specific value, and so on. However even if these features do exist in the source code, it does not necessarily follow that an extreme execution time will not be found. Therefore, Gross conducted an empirical study based on a handful of test objects to establish a system which could predict the testability of test objects, based on their source code. However, the empirical study was very small, consisting of only fifteen test objects. The type of test objects used was not characterized in any particular way, and the effects of the underlying hardware were not accounted for. Furthermore, the dependence of the prediction system on the setting of the genetic algorithm parameters was not established.

Wegener *et al.* [WPS99] investigated the objective function landscape for timing behaviour. They found that due to the fact execution times for several input vectors that execute the same program path can be identical, plateaux are common in the landscape. Discontinuities were also formed by significant differences in execution time for slightly different input vectors leading to the execution of different paths. These findings help explain why little improvement could be obtained by using local search to improve times found by genetic algorithms in the work of Wegener *et al.* [WSJE97] and Tracey [Tra00].

The experiments performed show the superiority of search-based approaches over random testing. Whilst search-based techniques cannot guarantee that the actual WCET or BCET will be found, the best result obtained can be used to form an interval with the time obtained from static analysis within which the *actual* extreme execution time most probably lies.

2.7 Conclusions

This chapter has surveyed the application of metaheuristic search techniques to software test data generation.

Search-based test data generation approaches to functional testing have largely focused on seeking input situations which demonstrate that an implementation does not conform to its specification. Execution of the test object is monitored, with input data solutions rewarded on the basis of how close they were to discovering a failure, as decided using the specification. Grey-box test data generation approaches combine methods used in generating functional and structural test data.

For structural test data generation, metaheuristic dynamic approaches were compared against static techniques based on symbolic execution. Techniques using symbolic execution evaluate program code in order to build up a system of constraints describing the test goal. However, this is problematic in the presence of loops and in cases where computed storage locations need to be determined. Instead of trying to formulate a constraint system, dynamic approaches merely execute the program with some input, and examine the effects via some form of program instrumentation. This helps circumvent some problems associated with static techniques, since dynamic information - for example pointer locations - are known at run-time. Metaheuristic techniques are then used to search for test data. The use of a metaheuristic technique requires the definition of an objective function which “rewards” test data solutions on the basis of how close they were to fulfilling the required test goal. *Coverage-oriented* objective functions reward input data on the basis of the number of program structures executed. It was argued, however, that *structure-oriented* approaches represent a more successful strategy. This is because each individual uncovered structure receives specific attention in the form of an individual search. Each individual search is provided explicit guidance to the coverage of the structure in question by an automatically tailored objective function. Without this guidance, nested structures only executed under special circumstances are unlikely to be exercized.

The techniques described for generating structural test data have focused on generating inputs for atomic function calls. The next chapter looks at the problem of this thesis - the generation of test data for test objects with state behaviour.

Chapter 3

The State Problem

3.1 Introduction

The last chapter showed how evolutionary algorithms can be successfully applied to the problem of automatically generating structural test data. The evolutionary approach has been shown to consistently achieve higher levels of coverage than random testing for a number of test objects [WBS01]. The techniques encountered so far have worked to generate input data for functions with input-output behaviour. However functions and components at higher system levels can store internal data, and can exhibit different behaviours based on the state of that data. The existence of state behaviour in test objects presents new challenges for evolutionary test data generation. Certain structures may require the generation of input sequences. Furthermore, the state of the test object might be managed by internal variables such as flags, counters and enumeration variables, which can result in flat or coarse objective function landscapes. These problems are confirmed in a series of experiments which attempt to generate test data for a series of state-based test objects using a standard evolutionary test data generation approach, simply referred to as the *standard evolutionary approach*. The literature is consulted once more, in order to draw on some recent ideas with respect to evolutionary input sequence generation. A new method, referred to as the *sequence evolutionary approach* is developed for the generation of input sequences for test objects involving several callable functions, for example as part of a program module.

3.2 The Challenges Caused by States in Test Objects

States in test objects present two major challenges for evolutionary structural test data generation:

3.2.1 Input Sequences

The standard evolutionary approach generates input vectors for single function calls. Test objects with states may require a sequence of calls to be generated in order for certain structures to be covered. This sequence may include calls to several different functions. Take the example of the C module representing a stack in Figure 3.1. In order to cover the statements that remove an element from the top of the stack - nodes b and c in the `pop` function - the `push` function needs to have first been called to put an element onto the stack, because initially, the stack is empty.

The "state" of the stack is managed by the `elements` array and the `size` counter variable. These "state variables" are declared using the `static` C keyword [KR88], which hides them from external calling processes. Therefore, the state of the stack can only be changed by invoking its visible functions. When the stack is empty, no calls to `pop` alone will lead to nodes b and c being executed.

Definition - State Variable
A state variable of a test object is an internal variable whose value is retained after the termination of a function of the test object until a function of the test object is next called.

3.2.2 The Internal Variable Problem

The use of internal variables in the conditions of programs can result in a degree of "information loss" when computing branch distance values, producing coarse or flat objective function landscapes for target structures within the program. This in turn results in the search receiving less guidance, making it less likely - if not impossible - that the required test data will be found. The degree of "difficulty" for the search depends on the level of information lost, which in turn depends on the type of the internal variable, and the form of assignments to it that appear in the program. Some internal variables only result in a small amount of information loss, which may not affect the success of the search.

The "`check_evaluations`" example

Take the "`check_evaluations`" function of Figure 3.2a. The test goal is to cover control flow graph node e, which requires the true branch to be taken from node d. The condition at node d depends on the internal variable `successful_evals`. However, the assignment at node b effectively renders the branch distance landscape for negative values of the input variable `evals`

```
CFG
Node
       static double elements[MAX_ELEMENTS];
       static int size = 0;

       double pop()
       {
 a         if (!empty())
           {
 b             size --;
 c             return elements[size];
           }
           else   // ...
       }

       void push(double d)
       {
 d         if (!full())
           {
 e             elements[size-1] = d;
 f             size ++;
           }
       }

       // ...
```

Figure 3.1: Fragment of code from a stack module

as flat (Figure 3.2b). In this portion of the search space, the search receives no guidance. The ideal branch distance landscape would be that of Figure 3.2c.

The "`count_event`" example

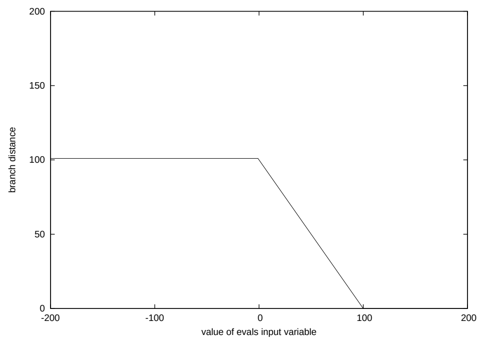
Evolutionary algorithms sample multiple points in the search space, and so it is likely that positive values of `evals` would be considered, meaning that the `check_evaluations` example should cause few problems in practice. But this is not always the case. Take the `count_event` function of Figure 3.3a. This example contains an internal state variable `counter`, declared using the `static` keyword again, meaning it will retain its value from one call of the function to the next. In order to cover node *e*, the true branch from node *d* must be taken. However, this branch predicate involves the variable `counter`, which is only incremented in a special case. The landscape for the true branch predicate is coarse (Figure 3.3b). No guidance is provided to the input values which will successively increment the `counter` variable at node *b*, and if the input domain to the function is relatively large, such an input sequence will not be found by chance. This means that the test object will never be brought into the required state for the target statement at node *e* to be executed.

The Stack example

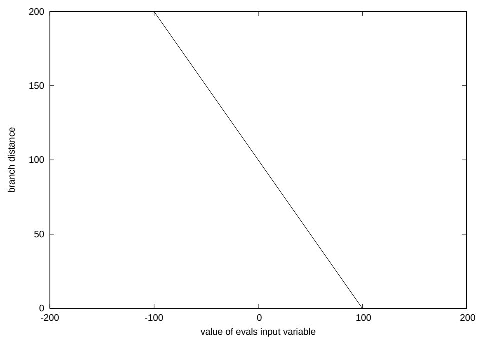
Flag and enumeration variables result in an almost complete loss of useful information at the branch predicate. The problems of flag and enumeration variables were introduced in Section 2.3.5. Consider the stack example again (Figure 3.1), and the situation where the false branch from node *d* must be covered, which requires the stack to be full. The predicate in the `if` statement at node *d* uses the return value from the `full` function, which returns a boolean value. The false branch distance landscape from node *d*, therefore, consists of two plateaux - one for all input sequences which do not lead to the evaluation of node *d* as false, and one for all input sequences which do lead to the evaluation of node *d* as false (Figure 3.4). An input sequence performing five "push" operations is deemed to be no closer to filling the stack than an input sequence performing only one "push". The search receives no guidance, and consequently becomes random. If a long input sequence is required to fill the stack, the chances of randomly finding a sequence that performs successive "push" operations in order for it to become full is very small. The search requires guidance so that successive executions of nodes *e* and *f* are performed, which increase the size of the stack, until eventually the stack is full.

```
CFG
Node
        const int THRESHOLD = 100;

        void check_evaluations(int evals)
        {
            int successful_evals = -1;
            int unsuccessful_evals = -1;

a           if (evals >= 0)
b               successful_evals = evals;
            else
c               unsuccessful_evals = -evals;

d           if (successful_evals > THRESHOLD)
e               // target statement

            // ...
        }
```

(a)



(b)

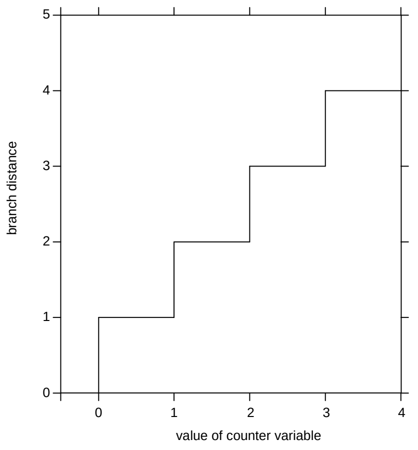


(c)

Figure 3.2: The `check_evaluations` example, demonstrating information loss with regards to branch distances based on an internal variable. (a) Program code. (b) Plot of true branch distance values from node 6, calculated using `|100 - successful_evals|`. (c) Plot of potential distances using the original input value, i.e. `|100 - evals|`

```
 CFG
 Node
          const int TARGET = 10;

          void count_event(int x)
          {
              static int counter = 0;

   a          if (x == 0)
   b              counter ++;
              else
   c              counter --;

   d          if (counter == TARGET)
   e              // target statement
          }
```

(a)



(b)

Figure 3.3: Information loss with regards to branch distances based on an internal counter state variable. a) Program code b) True branch distance plot for values of `counter` at node d, calculated using $|$`TARGET` $-$ `counter`$|$.

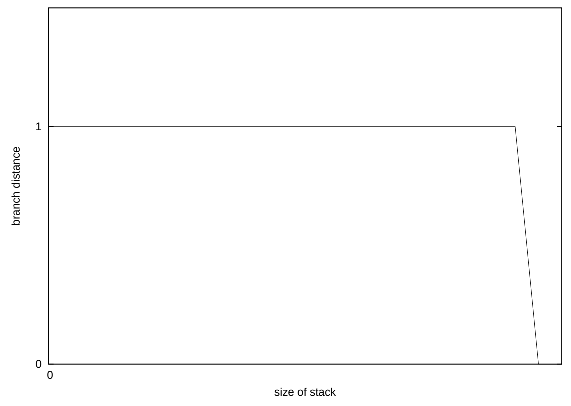


Figure 3.4: Branch distance plot using the return value of the `full()` function against size of the stack, calculated using $|1 - \texttt{full()}|$

The examples considered show how internal variables can cause problems for test objects with both input-output behaviour, through the `check_evaluations` example function, and state behaviour, through the `check_event` example function and the stack module. The use of internal variables is inevitable in state-based test object code, since they are required in order to manage the state. To make the problem worse, these variables are often of boolean flag nature, or of an enumeration type. Flag state variables can be used to indicate if the test object is in a particular state or not. Variables of an enumeration type can be used to indicate that the test object is in one of a set of states. Furthermore, flag variables are sometimes used as return values from auxiliary functions which query the state. Special enumeration values are also sometimes returned from auxiliary functions in order to denote the occurrence of some special event.

3.3 Experimental Study 1 - State-Based Experiments with the Standard Evolutionary Approach

This section performs experiments using a standard evolutionary approach for test objects with state-based behaviour. The aim is to show that the standard approach does indeed fail to address the challenges for test objects with states, as outlined in the previous section. The results obtained are compared with a random test data generation approach.

Table 3.1: Technical details of the synthetic state-based test objects

	Lines of code	Branches	Loops	Maximum Nesting Level	Functions (Public)
Anomaly Detector	59	14	2	3	3 (3)
Array Difference	31	12	2	3	1 (1)
Postcode	170	50	0	10	5 (1)
Sliding Window	127	24	3	3	10 (4)
Smoke Detector	40	14	0	2	1 (1)
Sortcode	97	26	0	4	4 (1)
Stack	51	8	0	1	5 (5)
Telephone Number	80	22	0	4	1 (1)
Vending Machine	112	26	4	4	5 (4)

3.3.1 Test Objects

A series of nine C programs were devised with state behaviour. Five test objects were composed of multiple functions as part of a module. These modules have internal state variables, global to each function, but hidden from manipulation by calling processes, as they are declared using the C `static` storage class [KR88]. The remaining four test objects consisted of a single function only. Each function contains several state variables declared internally to the function, again using the `static` storage class.

The test objects were designed to have varying levels of size and complexity, as can be seen in Table 3.1. The test objects also contain internal variables, including counters, flags and enumeration-type variables.

The source code for each test object can be found in Appendix B. Each test object was instrumented and prepared for the experiments as detailed in Section A.2.

Anomaly Detector

"Anomaly Detector" is a small module which monitors a stream of incoming data. Its purpose is to determine whether values (of type `double`) entered via the `add_data` function are inconsistent with regards to a history of past entered values.

Inputted values are stored in an internal array, which acts as a circular buffer - overwriting the oldest elements with new ones when it becomes full. The buffer has a fixed size of forty elements. When it is full, the `normal_limits` function can begin monitoring. To check for the consistency of the last entered value, the variance is found of the values currently in the buffer. The function returns a false value if the last value entered falls outside of this range. A third function in the module, `reset`, erases the internal buffer.

The range of the double input to the `add_data` used in the experiments was `-1,000` to `1,000` with a precision of `0.001`.

Array Difference

This test object consists of one function, which takes an array of ten integers as a parameter. The function returns true if the current inputted array contains the same values as for the array inputted in the last call of the function. The ranges used for the integer array values in the experiments were `-10,000` to `10,000`.

Postcode

The purpose of this test object is to check whether a sequence of unicode characters is a UK postcode of the form described by the regular expression $[A-Z][0-9]\{1,2\}[:blank:][0-9][A-Z][A-Z]$.

The test object consists of a principal function, which accepts unicode characters one at a time, with several internal auxiliary functions which return true or false values to indicate whether the character is a letter, a digit, a space, or a line feed. The line feed character signals the end of the postcode. The principal function has a high level of nesting, maintaining a small state machine through the use of internal flag variables, which keep track of how far through the validation process the system is. One of three constant integer values are returned. `RESULT_ENTER_NEW_CHAR` signals the system is ready to accept a new character; `RESULT_VALID` signals the entered sequence is valid, and that the system is ready to read in another postcode; or `RESULT_INVALID`, which signals that the last character was invalid.

The unicode character input to the function is modelled by an integer in the range `0` to `65,535`.

Sliding Window

This module is an implementation of the sliding window network protocol for reliable and efficient network data transmission. The code is adapted from and documented in Peterson and Davie [PD00].

Large data chunks are split into data "frames" for transmission across a network. In order to know if a particular frame needs to be re-transmitted, the sending side of the transmission needs to be informed that frames have successfully arrived at the receiver by means of an acknowledgment frame, which the receiver transmits back to the sender. In order to avoid an inefficient "stop-and-wait" approach to the receiving and acknowledgment of frames, the idea behind the sliding window protocol is that the sending and receiving sides both keep a respective window of currently acknowledged and received frames. Frames are numbered consecutively. The sender keeps an identification number of the next expected acknowledgment, whilst the receiver keeps the identification number of the next expected frame. When the sender receives an acknowledgment from the receiver that corresponds to a frame in the sending window, the sending window is advanced. When the receiver receives a frame in the receiving window, the receiving window is advanced.

The "Sliding Window" test object is a module which models the actions of both the sending and receiving side of a transmission. The three main functions in the module are `send`, `receive_ack` and `receive_frame`. The function `send` sends a frame, providing the send window is not full. The function `receive_ack` receives an acknowledgement for a sent frame, and advances the sending window if the acknowledgement denotes successful delivery of frames already in the window. The function `receive_frame` receives a frame on the receiving side and advances the receiving window. Another function, `reset`, resets the system. The module uses a series of internal functions, `send_window_not_full`, `is_in_window` and `is_next_frame` which check the state of the system and return boolean values.

The functions `send` and `receive_frame` take an array of integers as a parameter, which correspond to the message to be sent. For the experiments, the array size was `20` and each integer had a range `-10,000` to `10,000`. The function `receive_frame` also takes an integer parameter denoting the identification number of the frame received, which had a range `0` to `50,000`. The same range was also used for the integer parameter denoting the frame acknowledged for the function `receive_ack`.

Smoke Detector

This test object consists of one function which models a small controller for a smoke detector. The function takes one double input argument - the current room smoke level - followed by two flag output arguments used to signal whether the alarm should be switched on or off. The alarm works on a latch, so that after an "on" signal is received, the alarm will stay on until an "off" signal is received. The function is designed to be called repeatedly by the underlying hardware. When the room smoke level becomes higher than a given threshold for a certain period of time, the alarm is raised. When the room smoke level returns to safe levels for a given time, a special `waiting` flag variable becomes true. The alarm then stays on for another three seconds, unless the smoke levels breach acceptable limits again.

The range of the double input argument was `0` to `100` with a precision of 0.001.

Sortcode

This test object validates a UK bank sortcode of the form $XX - XX - XX$ where X is an integer digit. Unicode characters are submitted to a principal function one at a time. The principal function then uses a series of internal auxiliary functions which check whether a character is a digit, a hyphen or line feed. The line feed character signals the end of the inputted sortcode. Like the "Postcode" test object, the principal function acts as a small state machine, using boolean flags to denote the various stages through the validation process (for example one digit inputted, two digits inputted, and so on).

The unicode character input to the function is modelled by an integer in the range `0` to `65,535`.

Stack

This module implements a stack which can hold up to forty double values. The module implements `push`, `pop`, `check_size` and `reset` functions. The module uses an `error` variable to denote the fact that underflow or overflow errors have occurred, i.e. that the `pop` operation was invocated on an empty stack, or that the push operation has occurred on a full stack.

The range of the double parameter to the `push` function was `-10,000` to `10,000` with a precision of `0.001`.

Telephone Number

This function validates a UK telephone number. Numbers can be of international format (i.e. beginning with '0044' and consisting of 14 digits), national format (i.e. beginning with a zero and consisting of 11 digits) or local format (consisting of 7 digits and not beginning with a zero).

The test object attempts to perform the validation on a series of unicode characters entered into the principal function one at a time. No auxiliary functions are used, but a counter variable and several boolean variables are used to control the state of the validation.

The unicode character input to the function is modelled by an integer in the range `0` to `65,535`.

Vending Machine

This module simulates a vending machine which vends a small number of products. Several internal counter variables make up the internal state of the machine - namely the quantity of each product available for sale, and the quantity of each type of coinage left in the machine. Operations exist to buy products, enter money and retrieve change.

The function `insert_coinage` takes two integers, the first denoting the type of coinage and the second denoting the amount of coinage. The range for each of these parameters in the experiments was `0` to `10`. The function `buy_item` accepts an integer denoting an identification number for the product. The range used was `0` to `3`.

3.3.2 Experimental Setup

Full branch coverage was attempted for each of the test objects. Each branch is taken as the individual target of the search, regardless of whether it was fortuitously covered during the search for test data for another branch. The search for test data for each branch is repeated ten times. Experiments were performed on a Pentium 4 PC running Windows XP, with 3GHz and 1Gb RAM under normal load conditions. For further information on the technical details of the experimental framework, see Appendix A.

Setup for the Standard Evolutionary Approach

The standard evolutionary approach is a state of the art structure-oriented approach (as described in Section 2.3.5), using a objective function which combines both approach level and branch distance information, incorporating Equations 2.1, 2.2 and 2.3.

The evolutionary search generates inputs for the function containing the current structural target. A vector of floating point and integer variable values corresponding to the input data are optimized. The ranges and precision of each variable are specified. The test object is then called with this input data. The test object is reset to its initial state at the beginning of every generation of the search. The evolutionary search parameters used are documented in Section A.4, with 300 individuals per generation.

Should no test data be found for a particular branch, termination criteria are required to decide when the search should declare failure. For the experiments, two different stopping criteria were used. The **StdEA(200)** setup terminates the search after 200 generations. The **StdEA(1000)** setup terminates the search after 1000 generations. "StdEA" stands for *St*an*d*ard *E*volutionary *A*pproach.

Setup for Random Test Data Generation

The random approach simply generates inputs randomly for the function containing the current structural target. In order to mirror the evolutionary experiments, which resets the test object after every generation, the test object is reset after every 300th call. The maximum number of random evaluations that can be used for each branch is equivalent to the maximum number of evaluations used by the standard evolutionary approach searches for that branch (taking into account both successful and unsuccessful searches). The **Rnd(200)** setup, therefore, declares failure for a branch after a number of evaluations that is equivalent to the highest number of evaluations used for that branch by successful or unsuccessful searches of the StdEA(200) setup. The **Rnd(1000)** setup declares failure for a branch after a number of evaluations that is equivalent to the highest number of evaluations used for that branch by successful or unsuccessful searches of the StdEA(1000) setup.

Experimental Measurements

For each test object and test setup, the following information was measured:

- Success rate
- Coverage
- Average number of test data evaluations for a successful search
- Average number of test data evaluations for an unsuccessful search
- Maximum number of test data evaluations for a successful search
- Maximum number of test data evaluations for an unsuccessful search
- Average time for a successful search
- Average time for an unsuccessful search
- Maximum time for a successful search
- Maximum time for an unsuccessful search

Success rate is measured as follows:

$$success\ rate = \frac{100 \times successful\ searches}{r \times number\ of\ branches}$$

where $r = 10$ is the number of search repetitions for each test object branch and test setup.

Coverage represents the percentage of branches for which at least one of the ten searches for the test method was successful. Search times were measured with a precision of 0.1 of a second.

3.3.3 Results

Table 3.2 shows the coverage levels possible using both the standard evolutionary approach and random test data generation. Full coverage was not achieved for eight of the nine test objects with the standard evolutionary approach, whilst the random approach failed to achieve full coverage in all cases. The success rate for the standard evolutionary approach is also generally higher than for the random approach, as also shown in Table 3.2. An exception to this was the "Postcode" test object.

The number of evaluations for successful and unsuccessful searches are shown in Table 3.3. The number of test data evaluations performed by the standard evolutionary approach is always less than the number of individuals per generation multiplied by the maximum number of generations, as the objective values of parents reinserted into the next generation are retained. Increasing the number of evaluations available to each method (through an increase in the number of generations for the standard evolutionary approach) only resulted in a slight improvement on coverage and success rate.

Table 3.4 shows search times. The standard evolutionary approach takes longer to perform a similar number of evaluations as the random approach, due to the extra complexities of creating each successive generation, and interacting with the evolutionary algorithm server over a socket connection. Appendix A describes the details of the experimental framework.

Anomaly Detector

Full coverage could not be obtained with the "Anomaly Detector" test object, since all branches nested within an outer branch of the `normal_limits` function require the circular buffer to be full. This is not possible unless the call to the function is preceded by function calls to `add_data` to fill the buffer up. However, the standard evolutionary approach attempts to generate test data for the function containing the target structure only, ignoring other functions in the module.

Array Difference

Full coverage was obtainable with the “Array Difference” test object, even though the test object has internal states. A sequence length of up to two function calls is required in order for each branch to be covered. As the test object is not reset until the end of a generation, such a sequence was possible within the evaluation of a generation.

Even though test data can be generated in this manner, the resulting test data sequence might be very long. The branch could be covered in the last evaluation of the generation, resulting in an overly long test sequence where the majority of calls do not contribute towards coverage of the structural target in question. If the results of the test are checked manually, such long sequences would be a headache for the human checking the results of the test.

Postcode

The standard evolutionary approach exhibited very poor coverage with the “Postcode” test object. A sequence of calls is required to cover branches not corresponding to the initial state. This sequence of calls is unlikely to occur by chance during the evaluation of a generation of test data individuals.

Sliding Window

The search for test data with the “Sliding Window” test object suffered similar problems to “Anomaly Detector”, in that a function call sequence involving calls to other functions needed to have taken place in order for certain branches to be coverable.

Smoke Detector

Full coverage was not achieved with the “Smoke Detector” test object, because the required state could not be reached for a handful of branches. In order to reach the desired state a certain series of statements needed to be executed. However, the probability of this occurring is very small, unless specific guidance is given to the search. For example, in order to execute the branch where the alarm is switched off, the input value (the room smoke level) must be a very small value during the last six calls to the function. It is therefore unlikely that

Table 3.2: Success rate and coverage for the experiments using the standard evolutionary approach and random test data generation

Test Object	**Test Method**	**Success Rate (%)**	**Coverage (%)**
Anomaly Detector	StdEA(200)	21.4	21.4
	Rnd(200)	21.4	21.4
	StdEA(1000)	21.4	21.4
	Rnd(1000)	21.4	21.4
Array Difference	StdEA(200)	100.0	100.0
	Rnd(200)	91.7	91.7
	StdEA(1000)	100.0	100.0
	Rnd(1000)	91.7	91.7
Postcode	StdEA(200)	13.9	40.9
	Rnd(200)	24.8	37.9
	StdEA(1000)	20.5	53.0
	Rnd(1000)	37.4	39.4
Sliding Window	StdEA(200)	63.5	84.6
	Rnd(200)	53.5	80.8
	StdEA(1000)	71.5	84.6
	Rnd(1000)	68.1	80.8
Smoke Detector	StdEA(200)	84.3	85.7
	Rnd(200)	71.4	71.4
	StdEA(1000)	85.7	85.7
	Rnd(1000)	72.1	78.6
Sortcode	StdEA(200)	89.2	92.3
	Rnd(200)	66.9	84.6
	StdEA(1000)	89.2	92.3
	Rnd(1000)	81.9	84.6
Stack	StdEA(200)	75.0	75.0
	Rnd(200)	75.0	75.0
	StdEA(1000)	75.0	75.0
	Rnd(1000)	75.0	75.0
Tel No	StdEA(200)	73.8	85.3
	Rnd(200)	20.3	32.4
	StdEA(1000)	78.2	85.3
	Rnd(1000)	20.6	44.1
Vending Machine	StdEA(200)	71.9	71.9
	Rnd(200)	71.9	71.9
	StdEA(1000)	71.9	71.9
	Rnd(1000)	71.9	71.9

Table 3.3: Evaluations for the experiments using the standard evolutionary approach and random test data generation

Test Object	Test Method	Successful Search		Unsuccessful Search	
		Av Evals	Max Evals	Av Evals	Max Evals
Anomaly Detector	StdEA(200)	14	40	1	1
	Rnd(200)	14	40	1	1
	StdEA(1000)	14	40	1	1
	Rnd(1000)	14	40	1	1
Array Difference	StdEA(200)	146	3,550	-	-
	Rnd(200)	134	2,125	3,550	3,550
	StdEA(1000)	192	4,796	-	-
	Rnd(1000)	90	3,481	4,796	4,796
Postcode	StdEA(200)	8,633	51,085	53,010	54,943
	Rnd(200)	12,899	52,143	54,943	54,943
	StdEA(1000)	44,229	253,918	269,716	277,526
	Rnd(1000)	44,668	276,869	277,526	277,526
Sliding Window	StdEA(200)	1,929	47,615	53,554	54,859
	Rnd(200)	4,540	36,714	48,882	54,859
	StdEA(1000)	21,051	260,147	267,154	268,205
	Rnd(1000)	20,661	225,142	217,720	268,205
Smoke Detector	StdEA(200)	191	3,011	53,477	53,485
	Rnd(200)	3	4	42,482	53,485
	StdEA(1000)	1,053	97,357	267,077	267,085
	Rnd(1000)	3	56	162,131	267,085
Sortcode	StdEA(200)	3,111	36,411	53,934	58,681
	Rnd(200)	3,527	42,932	30,848	58,681
	StdEA(1000)	6,719	151,216	268,495	277,100
	Rnd(1000)	11,222	252,092	126,196	277,100
Stack	StdEA(200)	15	42	1	1
	Rnd(200)	15	42	1	1
	StdEA(1000)	15	42	1	1
	Rnd(1000)	15	42	1	1
Tel No	StdEA(200)	3,842	45,454	56,088	57,066
	Rnd(200)	1,853	21,371	31,020	57,066
	StdEA(1000)	9,583	239,054	272,032	277,613
	Rnd(1000)	2,608	27,040	142,025	277,613
Vending Machine	StdEA(200)	4	37	17,826	53,500
	Rnd(200)	5	71	53,500	53,500
	StdEA(1000)	4	38	89,050	267,173
	Rnd(1000)	4	47	267,173	267,173

Table 3.4: Search times for the experiments using the standard evolutionary approach and random test data generation

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Av Time (s)	**Max Time (s)**	**Av Time (s)**	**Max Time (s)**
Anomaly Detector	StdEA(200)	0.3	0.4	0.1	0.1
	Rnd(200)	0.1	0.1	0.1	0.1
	StdEA(1000)	0.3	0.4	0.1	0.1
	Rnd(1000)	0.1	0.1	0.1	0.1
Array Difference	StdEA(200)	0.5	2.3	-	-
	Rnd(200)	0.2	0.4	0.7	0.7
	StdEA(1000)	0.6	2.9	-	-
	Rnd(1000)	0.1	0.7	0.8	0.8
Postcode	StdEA(200)	1.6	8.2	8.3	9.2
	Rnd(200)	0.1	1.0	1.2	1.2
	StdEA(1000)	7.7	46.6	49.9	54.9
	Rnd(1000)	0.8	6.3	6.6	6.7
Sliding Window	StdEA(200)	0.8	11.9	12.3	13.6
	Rnd(200)	2.3	3.9	4.0	4.6
	StdEA(1000)	6.1	72.2	63.9	76.6
	Rnd(1000)	12.1	16.5	16.4	16.8
Smoke Detector	StdEA(200)	0.4	0.9	11.8	13.1
	Rnd(200)	0.1	0.1	8.1	15.9
	StdEA(1000)	0.6	21.1	68.4	69.7
	Rnd(1000)	0.1	0.1	4.2	7.0
Sortcode	StdEA(200)	1.0	7.9	11.7	12.1
	Rnd(200)	0.1	1.2	0.9	2.1
	StdEA(1000)	1.8	35.6	67.2	68.9
	Rnd(1000)	0.3	6.7	6.8	6.9
Stack	StdEA(200)	0.3	0.4	0.1	0.1
	Rnd(200)	0.1	0.1	0.1	0.1
	StdEA(1000)	0.3	0.4	0.1	0.1
	Rnd(1000)	0.1	0.1	0.1	0.1
Tel No	StdEA(200)	1.0	9.5	10.2	11.1
	Rnd(200)	0.1	0.4	0.6	1.7
	StdEA(1000)	2.3	51.3	59.3	60.8
	Rnd(1000)	0.1	0.5	2.2	7.9
Vending Machine	StdEA(200)	0.3	0.6	2.9	10.0
	Rnd(200)	0.1	0.1	3.1	3.3
	StdEA(1000)	0.3	0.5	17.1	54.4
	Rnd(1000)	0.1	0.1	14.2	14.3

the standard evolutionary approach will develop such a sequence by chance during the evaluation of a generation of test data individuals.

Sortcode

Full coverage was not obtainable for the “Sortcode” test object. Although the test object consisted of one principal callable function, some branches required the test object to be in a certain state for which an input sequence was unlikely to be developed during a generation of test data individuals.

Stack

Testing with the “Stack” module suffered from similar problems to the “Anomaly Detector” and “Sliding Window” modules in that other functions needed to be executed to put the test object into some state for branches in the current function under test to be coverable. In the case of the Stack module, elements could not be pushed onto the stack in order for branches in the “pop” function to be feasible.

Telephone Number

Full coverage was not obtainable for the “Telephone Number” test object. Again, the test object consisted of one principal callable function, but some branches required the test object to be in certain states for which an input sequence was unlikely to be developed during a generation of test data individuals.

Vending Machine

The “Vending Machine” module required input sequences to be developed for certain structures which required other functions in the module to have been invocated. Such a sequence cannot be developed during a generation of test data individuals, which correspond to atomic calls to the function containing the target structure only.

3.3.4 Conclusions of Experimental Study 1

The experiments show that it is not always possible for the standard evolutionary approach to test data generation to achieve full coverage for test objects with state behaviour. Function call sequences are often required in order to put internal variables into a certain state for some structures to be coverable. The usual approach to evolutionary structural testing does not evolve function call sequences, but test data for single calls. If the test object is reset at the end of each generation, it is possible for the state to be changed within the evaluation of that generation, resulting in the coverage of some state-dependent structures. However, this is not a desirable strategy as the resulting function call sequence can be very long. Different functions in a test object may be involved in changing the state, for example with modules or abstract data types. Since the standard evolutionary approach only generates test data for the function which contains the current structure of interest, the method will not invoke these functions, leaving these structures uncovered.

3.4 Relevant Literature for Evolutionary Input Sequence Generation

This section delves into the literature again to analyse works involving evolutionary input sequence generation. The work of Baresel *et al.* [BPS03] evolves test data sequences for single function test objects with internal states, whilst recent work by Tonella [Ton04] involves the use of evolutionary algorithms to generate sequences of constructor and method calls for structural testing of object-oriented classes.

3.4.1 Generation of Input Sequences for Single Function Test Objects with State Behaviour

Baresel *et al.* [BPS03] generate input sequences for test objects consisting of a single function with state behaviour. Individuals are encoded as sequences of input vectors to the function. The sequence is of length l, in order for l calls to the function to be performed. Since the function is now called many times, the individual has many chances to execute the desired branch. The objective value of the individual is calculated using the approach level and normalized branch distance at the closest point of executing the branch:

$$min(approach\_level + normalize\_bd(branch\_distance)) \qquad (3.1)$$

Take the example of Figure 3.5 and the input sequence:

```
<(6, 6, 5), (2, 2, 3), (2, 2, 3), (6, 6, 4), (6, 6, 5)>
```

The individual is closest to executing the true branch of branching node in the last call, where the branch distance is $5-3+K = 2+K$ (using Tracey's functions

```
const int THRESHOLD = 5;

int sequence_example(int a, int b, int c)
{
    static int counter = 0;
    if (((a + b) / 2) > c)    // branching node 1
        counter ++;

    if (counter >= THRESHOLD)  // branching node 2
        return 1;

    return 0;
}
```

Figure 3.5: Test sequence generation example

for calculating branch distances, shown in Table 2.2). It is not required that the individual must execute the target structure during the last call to the function.

One drawback to the scheme is that the tester must have some idea of how long the sequence is going to be, in order to set the value of l. If l is too low, the generated sequences might be too short, leaving some target structures uncovered. If l is too high, the generated sequences will be very long, meaning the search may take longer or fail to find test data because it is operating in a larger search space.

3.4.2 Evolutionary Structural Testing of Classes

The work of Tonella [Ton04] involves the use of evolutionary algorithms to automate the structural testing of classes in object-oriented Java systems. The state problem exists in the testing of object-oriented systems, since objects store internal data in "member variables", which are manipulated by the methods of the object.

The steps required for the coverage of some structure contained in a class are as follows:

1. The creation of an object of the class, using one of its constructors;
2. The invocation of a sequence of methods to bring the object into the required state;
3. Invocation of the method containing the target structure [Ton04]

Therefore, the encoding used by Tonella describes a sequence of constructor and method calls, with associated input data for those calls. If an input is one of the basic Java types, for example an integer, a double or a string, it is randomly generated and manipulated by the evolutionary algorithm. If the input is an object, the object must be instantiated as part of the sequence. Further method calls are allowable on the object if it has to be put into some desired state.

Care is taken with the mutation and crossover operators so that the validity of the sequence is not destroyed (for example, the constructor of an object could be accidently removed, leaving method calls to a non-existent object). The mutation operators are allowed to perform one of the following actions:

- **Regeneration of an input value.** An input value of a Java basic type is replaced by a randomly generated value of the same type
- **Change of Constructor**. One of the constructors used in the sequence is swapped for another constructor available in the class. New inputs are generated for the constructor call. If the constructor takes an object as an input, an existing object created in the sequence may be reused, or, a new object is instantiated by the insertion of a new constructor call.
- **Insertion of method invocation**. A new method call is inserted into the sequence. New inputs are generated for the call. If the method takes an object as an input, an existing object created in the sequence may be reused, or, a new object is instantiated by the insertion of a new constructor call.
- **Removal of a method invocation**. A method is removed from the sequence, with its input values. If an object is used as an input and not used elsewhere in the sequence, the constructor and method calls to that object are also removed.

Recombination is performed by a one-point crossover operator. After recombination, unnecessary constructors and method calls are removed, with any missing constructor calls added.

Tonella's experiments concentrate on obtaining branch coverage for a handful of classes from the standard Java library - these being the `StringTokenizer`, `BitSet`, `HashMap`, `LinkedList`, `Stack` and `TreeSet` classes. The objective function used is similar to that used by Pargas *et al.* [PHP99] - rewarding individuals on the basis of the number of control dependent nodes executed en route to the target. No branch distance information is used. Even so, in the experiments good levels of coverage were achieved, with all but eleven branches covered. Five branches were infeasible. One branch in the `HashMap` class required the hash table to have a billion entries; whilst four could not be covered in the `BitSet` class because one method needed to be called with an object equal to the `this` object or with a parameter that of type `BitSet`.

3.5 An Approach for the Generation Input Sequences for Multiple Function Test Objects

This section directly extends the work of Baresel *et al.* [BPS03] to generate input sequences for procedural C test objects with multiple functions. This requires a modification to the encoding of individuals, which must now represent a function call sequence involving a possible number of functions, rather than just one. Other features of the method, such as the objective function definition (Equation 3.1), remain the same. The method is referred to as the *sequence evolutionary approach*.

3.5.1 Modified Encoding

The modified encoding consists of a "generic function call" sub-encoding which is repeated l times. The generic function call sub-encoding represents a possible call to any of the functions in the test object. It begins with a function identification number, followed by a "universal parameter vector" which maps in a different way to each function call signature of the test object.

An example of generating such an encoding for a test object can be seen in Figure 3.6. The universal parameter vector, and the mapping from each function signature to the vector, is constructed as follows.

First, positions are made in the vector which correspond to the arguments of the first function encountered. In the example, the parameters `i`, `j` and `k` map to the first three positions; which are reserved for double, integer and integer types respectively, corresponding to their types in this function. The construction algorithm then attempts to map the parameters of the remaining functions into this vector. Therefore, integer parameter `i` of function 2 maps into position 2. The double parameter `q` can map into position 1. A new position has to be added to the vector for the remaining integer argument - position 4.

When function signatures vary in terms of the number of variables of each different type there is some enforced redundancy. In the example, position 4 is redundant as far as function 1 is concerned, and position 3 is redundant for function 2.

Function signatures:

```
void function1(double i, int j, int k)
void function2(int p, double q, double r)
```

Generic Encoding for a Function Call:

Function ID	**Position 1** double argument (1)	**Position 2** integer argument (1)	**Position 3** integer argument (2)	**Position 4** double argument (2)

Mapping of Encoding to Function 1:

1	`i`	`j`	`k`	*ignored*

Mapping of Encoding to Function 2:

2	`q`	`p`	*ignored*	`r`

Figure 3.6: Generating a generic encoding and mapping the universal parameter set to individual function call signatures

When there is only one function in the test object, the function identification number can be removed, and the encoding becomes identical to that of the original scheme of Baresel *et al.*

The requirement to select a value of l could be dropped through the use of a variable length encoding. This is however, out of the scope of this thesis.

3.6 Experimental Study 2 - State-Based Experiments with an Evolutionary Structure-Oriented Sequence Generation Approach

3.6.1 Experimental Setup

Experiments were performed using the new sequence evolutionary approach with the same experimental test object set of experimental study 1. It was ensured that the fixed sequence lengths for generating the function call sequence encoding were long enough for all branches to be covered (Table 3.5). The test object was reset after the evaluation of each test data sequence. Each search for each branch and each of the different setup outlined below was repeated ten times.

The sequence evolutionary approach was performed in two variants, the first using 200 generations as a termination criterion, and the second 1000 generations. These setups are respectively referred to as the **SeqEA(200)** and

Table 3.5: Function call sequence lengths used for each state-based test object used during the sequence generation experiments

Test Object	**Function Call Sequence Length**
Anomaly Detector	45
Array Difference	2
Postcode	10
Sliding Window	6
Smoke Detector	10
Sortcode	10
Stack	45
Telephone Number	15
Vending Machine	5

SeqEA(1000) setups. "SeqEA" stands for *Seq*uence *E*volutionary *A*pproach.

As with experimental study 1, the evolutionary approach was compared with a random approach. This time, however, the random approach generates test sequences. Once more, two setups were used for the random approach, in order to mirror the evolutionary approach. The **SeqRnd(200)** setup declares failure for a branch after a number of evaluations that is equivalent to the highest number of evaluations used for that branch by successful or unsuccessful searches of the SeqEA(200) setup. The **SeqRnd(1000)** setup declares failure for a branch after a number of evaluations that is equivalent to the highest number of evaluations used for that branch by successful or unsuccessful searches of the SeqEA(1000) setup.

The same types of information were recorded as for experimental study 1.

3.6.2 Results

Table 3.6 shows that the sequence evolutionary approach achieves higher coverage levels and a higher success rate than the random sequence approach over a similar number of input sequence evaluations. Figure 3.8 compares success rate levels and Figure 3.7 compares coverage levels for the standard evolutionary approach against the sequence evolutionary approach. The sequence evolutionary approach achieves higher coverage levels and success rates in all cases apart from the "Array Difference" and "Telephone No" test objects.

Despite improving on the standard approach in most cases, the sequence approach still fails to achieve full coverage. The reasons for this are explained with respect to each test object below:

Table 3.6: Results for the experiments using the sequence evolutionary approach and random sequence approach

Test Object	**Test Method**	**Success Rate (%)**	**Coverage (%)**
Anomaly Detector	SeqEA(200)	21.4	21.4
	SeqRnd(200)	14.3	14.3
	SeqEA(1000)	21.4	21.4
	SeqRnd(1000)	14.3	14.3
Array Difference	SeqEA(200)	91.7	91.7
	SeqRnd(200)	91.7	91.7
	SeqEA(1000)	91.7	91.7
	SeqRnd(1000)	91.7	91.7
Postcode	SeqEA(200)	46.1	57.6
	SeqRnd(200)	17.7	25.8
	SeqEA(1000)	48.3	66.7
	SeqRnd(1000)	23.6	36.4
Sliding Window	SeqEA(200)	83.5	100.0
	SeqRnd(200)	66.9	84.6
	SeqEA(1000)	83.8	100.0
	SeqRnd(1000)	73.8	88.5
Smoke Detector	SeqEA(200)	86.4	92.9
	SeqRnd(200)	72.1	78.6
	SeqEA(1000)	87.1	92.9
	SeqRnd(1000)	72.9	78.6
Sortcode	SeqEA(200)	87.7	92.3
	SeqRnd(200)	73.5	88.5
	SeqEA(1000)	89.2	92.3
	SeqRnd(1000)	74.6	88.5
Stack	SeqEA(200)	87.5	87.5
	SeqRnd(200)	75.0	75.0
	SeqEA(1000)	87.5	87.5
	SeqRnd(1000)	75.0	75.0
Tel No	SeqEA(200)	60.6	76.5
	SeqRnd(200)	42.9	47.1
	SeqEA(1000)	63.8	79.4
	SeqRnd(1000)	43.2	44.1
Vending Machine	SeqEA(200)	96.6	96.9
	SeqRnd(200)	92.8	93.8
	SeqEA(1000)	96.9	96.9
	SeqRnd(1000)	92.8	96.9

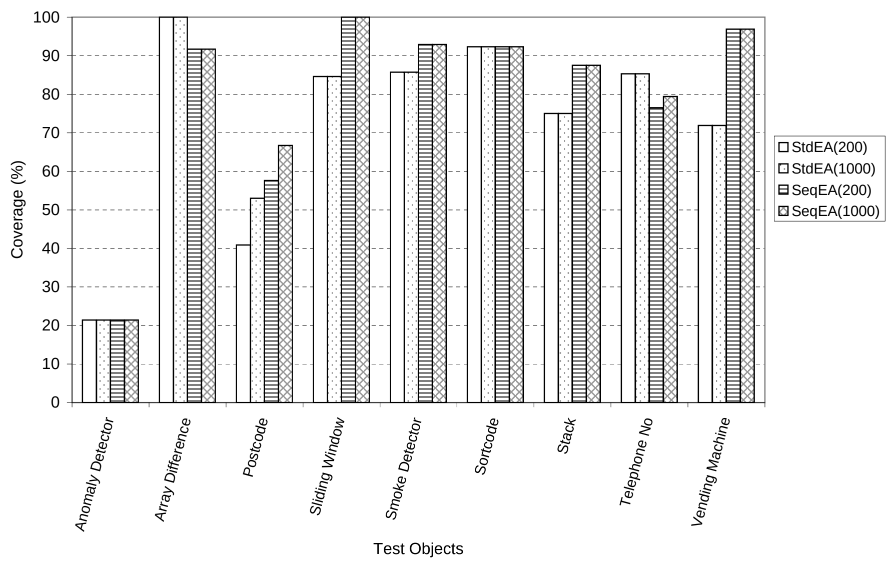


Figure 3.7: Comparison of coverage levels for the standard evolutionary approach verses the sequence evolutionary approach

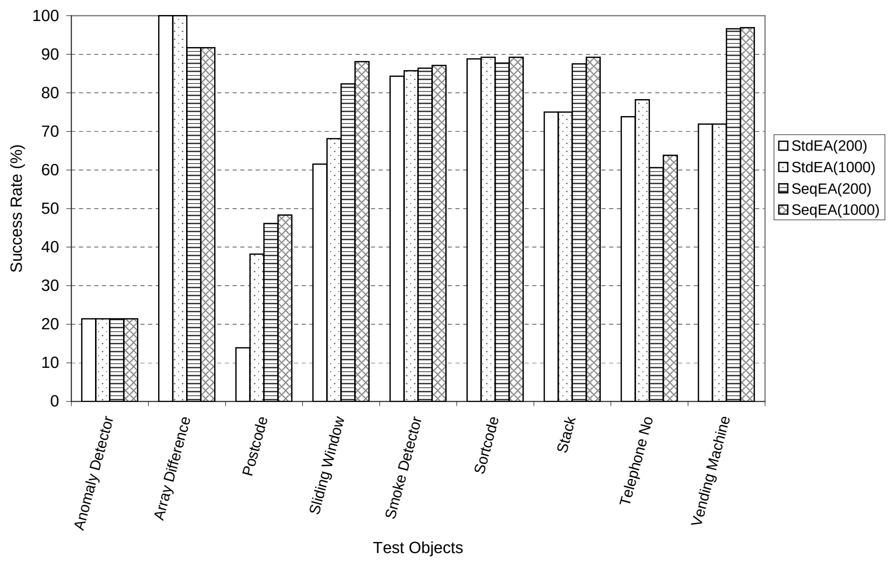


Figure 3.8: Comparison of success rate levels for the standard evolutionary approach verses the sequence evolutionary approach

Anomaly Detector

For the "Anomaly Detector" test object, the majority of branches are nested within an `if` condition that relies on the boolean state variable `buffer_full` being true. As the name of the variable suggests, this boolean flag is not true unless the buffer is at maximum capacity. However, due to the use of a flag to represent this state, the search receives no positive feedback when elements are added to the buffer. As the buffer is forty elements in size, the chances of developing a sequence which keeps calling the `add_data` function and filling the buffer are small, and the search fails.

Array Difference

The "Array Difference" test object produces unusual results, in that full coverage is obtained by the standard evolutionary approach, but not by the sequence evolutionary approach.

One branch requires that array inputs to subsequent calls of the function are identical. A flag variable is used to store the fact that this is the case. It was not expected that this branch would be covered by either technique. However in all runs for the standard evolutionary approach, the branch was covered, due to adjacent reinsertion of a parent and an identical offspring within a generation.

Postcode

The "Postcode" test object contains many internal boolean state variables, and uses a series of boolean return values from auxiliary functions. Another contributing fact was the high level of nesting in the program (refer to Section 2.3.5), which hinders the evolutionary search.

Sliding Window

The "Sliding Window" test object contains two functions that return boolean values - `is_in_window` and `is_next_frame` - which appear in three conditions in the program. The use of these flags prevented the search finding input data, due to a lack of search guidance.

Smoke Detector

The "Smoke Detector" test object uses three flags and two counter variables. Three branches are generally problematic because the `detected` counter needs to be reduced to zero. However this only occurs when the inputted smoke

Table 3.7: Evaluations for the sequence and random sequence generation experiments

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Av Evals	**Max Evals**	**Av Evals**	**Max Evals**
Anomaly Detector	SeqEA(200)	2,202	7,837	53,488	53,492
	SeqRnd(200)	1	1	49,686	53,492
	SeqEA(1000)	2,123	6,815	267,088	267,092
	SeqRnd(1000)	1	1	245,401	267,092
Array Difference	SeqEA(200)	571	12,347	53,487	53,487
	SeqRnd(200)	173	7,475	53,487	53,487
	SeqEA(1000)	404	6,755	267,087	267,087
	SeqRnd(1000)	75	2,792	267,087	267,087
Postcode	SeqEA(200)	6,731	46,011	56,781	57,984
	SeqRnd(200)	1,928	57,932	57,984	57,984
	SeqEA(1000)	18,954	255,413	277,437	279,506
	SeqRnd(1000)	35,192	238,757	279,506	279,506
Sliding Window	SeqEA(200)	6,898	52,941	53,483	53,511
	SeqRnd(200)	6,335	51,198	29,458	53,511
	SeqEA(1000)	10,003	194,258	267,119	267,198
	SeqRnd(1000)	17,296	234,975	46,462	267,198
Smoke Detector	SeqEA(200)	332	7,006	53,487	53,487
	SeqRnd(200)	2	49	33,860	53,487
	SeqEA(1000)	536	16,110	267,087	267,087
	SeqRnd(1000)	6	241	152,098	267,087
Sortcode	SeqEA(200)	2,395	52,953	55,292	57,163
	SeqRnd(200)	2,973	49,170	38,812	57,163
	SeqEA(1000)	2,937	146,675	269,160	272,569
	SeqRnd(1000)	1,992	32,133	184,047	272,569
Stack	SeqEA(200)	1,191	8,846	53,491	53,516
	SeqRnd(200)	1	1	31,181	53,516
	SeqEA(1000)	1,177	10,154	267,100	267,124
	SeqRnd(1000)	1	1	138,639	267,124
Tel No	SeqEA(200)	5,029	45,282	57,553	57,814
	SeqRnd(200)	1,320	35,709	57,814	57,814
	SeqEA(1000)	13,000	203,009	277,255	297,459
	SeqRnd(1000)	1,467	8,738	297,459	297,459
Vending Machine	SeqEA(200)	171	7,012	54,070	56,924
	SeqRnd(200)	9	192	49,338	56,924
	SeqEA(1000)	192.8	8,052	267,116	267,136
	SeqRnd(1000)	702	205,941	108,048	267,136

Table 3.8: Search times for the sequence and random sequence generation experiments

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Av Time (s)	**Max Time (s)**	**Av Time (s)**	**Max Time (s)**
Anomaly Detector	SeqEA(200)	4.2	13.1	72.8	74.6
	SeqRnd(200)	0.1	0.1	9.4	10.4
	SeqEA(1000)	4.1	11.5	379.2	391.3
	SeqRnd(1000)	0.1	0.1	46.5	51.5
Array Difference	SeqEA(200)	0.8	4.7	36.1	36.2
	SeqRnd(200)	0.1	1.1	7.8	7.8
	SeqEA(1000)	0.7	2.8	189.6	189.9
	SeqRnd(1000)	0.1	0.5	39.3	39.5
Postcode	SeqEA(200)	4.9	31.4	41.7	43.4
	SeqRnd(200)	0.3	11.3	13.4	13.5
	SeqEA(1000)	15.1	227.1	228.1	229.3
	SeqRnd(1000)	5.1	35.0	45.0	45.1
Sliding Window	SeqEA(200)	4.9	37.1	36.7	37.8
	SeqRnd(200)	0.3	2.8	3.3	4.5
	SeqEA(1000)	7.3	143.9	196.3	204.9
	SeqRnd(1000)	1.5	14.6	16.9	102.4
Smoke Detector	SeqEA(200)	1.0	8.8	61.9	62.3
	SeqRnd(200)	0.1	0.1	8.2	16.0
	SeqEA(1000)	1.2	19.4	316.2	318.1
	SeqRnd(1000)	0.1	0.1	37.2	52.0
Sortcode	SeqEA(200)	3.4	65.4	66.0	67.0
	SeqRnd(200)	0.6	4.3	2.5	4.5
	SeqEA(1000)	4.1	188.9	340.2	342.2
	SeqRnd(1000)	1.4	8.9	16.8	21.5
Stack	SeqEA(200)	2.6	14.8	71.4	76.4
	SeqRnd(200)	0.1	0.1	6.2	11.0
	SeqEA(1000)	2.6	17.1	362.3	377.0
	SeqRnd(1000)	0.1	0.1	26.9	54.4
Tel No	SeqEA(200)	5.4	48.5	52.0	52.5
	SeqRnd(200)	0.3	7.7	15.2	15.3
	SeqEA(1000)	13.8	246.7	272.0	292.2
	SeqRnd(1000)	0.4	2.0	67.7	67.8
Vending Machine	SeqEA(200)	0.5	3.4	24.5	24.9
	SeqRnd(200)	0.1	0.1	3.51	4.3
	SeqEA(1000)	0.5	3.9	139.4	143.0
	SeqRnd(1000)	0.1	14.5	7.7	19.2

Table 3.9: Earliest, latest and average generation of last individual offering improvement on the best objective value found in the sequence evolutionary approach test data searches for the branches of each test object

	SeqEA(200)			**SeqEA(1000)**		
	Earliest	Latest	Average	Earliest	Latest	Average
Anomaly Detector	1	1	1	1	1	1
Array Difference	1	1	1	1	1	1
Postcode	1	199	45	1	966	101.7
Sliding Window	1	1	1	1	1	1
Smoke Detector	1	48	7.3	1	146	9.1
Sortcode	1	44	10.5	1	53	7.4
Stack	1	1	1	1	1	1
Tel No	1	192	31.4	1	777	62.7
Vending Machine	1	24	3.1	1	1	1

level value is very small, and the search receives no explicit guidance to these statements.

Sortcode

The “Sortcode” test object uses several internal boolean variables to manage the state, as well as using boolean values as return types from auxiliary functions. The use of these boolean values prevented the search from receiving adequate guidance, failing to find the required input sequences.

Stack

“Stack” uses an auxiliary function which returns an error code corresponding to error states like underflow or overflow. The overflow error code causes a problem because it appears in a condition testing if the stack is full before pushing an element onto the stack. However the error variable values have a similar effect on the search as values from an enumeration, with plateaux forming on the objective function landscape. Guidance is not provided to the search as to how the stack becomes full, and the search fails.

Telephone No

The “Telephone” test object uses several internal boolean variables to manage the state. Once again, the use of these variables led to objective function landscapes that provided inadequate guidance to the search, leading to failure to find the required input sequences and preventing full coverage.

Vending Machine

Full coverage would have been obtained for the "Vending Machine", but for the use of an auxiliary function returning a boolean value, which prevented coverage of one of its branches.

Search Times and Evaluations

Evaluations and search times can be seen in Tables 3.7 and 3.8 respectively. The maximal number of test data evaluations performed by the sequence evolutionary approach is less than the number of individuals per generation multiplied by the maximal number of generations, as the objective values of parents reinserted into the next generation are retained.

Increasing the number of evaluations available to each approach (i.e. through the increased number of generations as the termination criterion for SeqEA(1000) only resulted in an improvement of coverage levels for two test objects - "Postcode" and "Telephone Number". Only a slight improvement in success rate was obtained for six test objects, with no improvement for the remaining three.

Table 3.9 shows that on average, unsuccessful searches as early as the 100th generation, for both SeqEA(200) and SeqEA(1000) setups. This suggests that the results are unlikely to be significantly improved by simply running the searches for even longer than 1000 generations.

3.6.3 Conclusions of Experimental Study 2

Experimental study 2 shows that sequences can be generated for procedural C test objects containing multiple functions, through the use of evolutionary algorithms. The evolutionary approaches always outperforms the random method.

The sequence evolutionary approach improves upon the standard evolutionary approach in the majority of cases in terms of both coverage levels and search success rate. Furthermore, the sequence approach can guarantee the lengths of generated input sequences are of a reasonable size, as determined by the tester. For the standard approach, test data for covered branches corresponds to the sequence formed from the first individual in the generation up to the individual on which the branch was executed.

The sequence evolutionary approach was only able to obtain full coverage for one of the test objects. Full coverage was prevented by the lack of guidance provided to the search, as a result of the use of internal variables such as counters and boolean flags.

3.7 Conclusions

This chapter showed how states in test objects can cause problems for the standard structure-oriented approach to evolutionary structural testing (referred to as the *standard evolutionary approach*), which was designed to generate test data for single function calls. However the existence of states in test objects requires that function call sequences are generated in order for certain structures to be covered. The issue of function call sequence generation has been addressed in the literature. The work of Baresel *et al.* [BPS03] generates sequences for test objects with one function. Tonella [Ton04] describes an approach for the structural testing of classes. The work of Baresel *et al.* was extended in order to handle multiple function test objects in a method referred to as the *sequence evolutionary approach*. Whilst the sequence approach generally obtains higher coverage than the standard approach, full coverage could still not be obtained in a number of cases. This is due to use of internal variables in the conditions of programs. Internal variables can result in a degree of "information loss" when computing branch distance values, producing coarse or flat objective function landscapes for structures within the program. This in turn results in the search receiving less guidance to the required test data, leading to possible search failure. The use of internal variables within program code for test objects with state behaviour is inevitable, as internal variables are required to manage the state. Somehow, the search needs to be provided with extra guidance so that the internal variable problem can be overcome. This is the subject of the next chapter.

Chapter 4

Revisiting the Problem of Internal Variables

4.1 Introduction

The problem of internal variables has already been encountered in the literature in the form of flag variables. However, other types of internal variables can also hinder the finding of test data if they cause the objective function landscape to become flat or coarse, offering insufficient guidance to the search. The use of internal variables is inevitable for test objects with state behaviour, since internal state variables of some form or other are required in order to manage the current state.

This chapter looks into the literature for possible inspiration for solving the problem, beginning with techniques used to deal with the flag problem. These methods are investigated to see if they can be extended to the more general problem. Following this, the possibility of reverse engineering state machine models of the system is discussed. A state model may provide insight into the internal workings of a test object, providing information that could be used to guide the test object into certain states required for certain structures to be covered.

A previously unconsidered idea is the application of Korel and Ferguson's chaining approach. The basic idea of the chaining approach is to identify a sequence of statements involving assignments to internal variables, which might need to be executed prior to the structural target in question. By requiring these statements to be executed, further information can be made available for guidance of the search, which may improve the chances of finding test data.

4.2 Solutions to the Flag Problem

The flag problem was originally discussed in this thesis in Section 2.3.5. Solutions to the flag problem are now examined to see if they can be extended to the more general problem of internal variables.

The approach of Baresel *et al.* [BS03] statically analyses program code to identify flags within the code. Statements assigning the desired value to the flag are found, and extra guidance is provided via the objective function so that these statements are executed. The testability transformation approach of Harman *et al.* [HHH+02, HHH+04] uses static analysis to transform flags out of the conditions of the test object's code.

The main problem with extending these approaches is that the form of objective function landscape for a structure must be predictable from the program code prior to the search being undertaken. This is straightforward for flag variables, where it is known that the landscape will be flat, preventing the search from receiving guidance. For other types of internal variable, the shape of the landscape cannot always be forecast in advance.

4.3 Use of a State-based Model of the System

A state-based model of a system may reveal information about the internal workings of the test object so that extra guidance can be provided to the search where it is needed; for example to put the test object in the required state for certain target structures to be coverable, via certain assignments to internal state variables.

One source of a state-based model would be a state-based formal specification of the system produced as part of the development process. However, not all systems are engineered this way. Even if a state-based specification did exist, it would more than likely be too abstract to reveal source level details for structural testing of the system. Another possibility is to reverse engineer the required information from the code itself, and use this information to assist the search. In the literature, state models have been reverse engineered from the source code of a system to assist in testing and model-checking.

4.3.1 Reverse Engineering State Models for Testing

Kung *et al.* [KSGH94, KLV+96] reverse state models from program code for state-based testing of objects.

The state space of an object is the product of the domains of its internal member variables. Since the different values of member variables lead to different paths being taken through the object's different methods, different behaviour arises depending on the state. The product of member variable domains can potentially be extremely large. However, this space can be partitioned into a smaller set of states on the basis that paths can be executed by more than one value from a member variable's domain.

Kung *et al.* drive this partitioning through the use of symbolic execution (introduced in Chapter 2). The resulting state model is called the "object state diagram". The technique can be demonstrated using the example of the `CoinBox` class [KSGH94], seen in Figure 4.1, which models a simple vending machine. The class has methods to add and retrieve money (`AddQtrs()` and `ReturnQtrs()`), vend products (`Vend()`) and reset the machine (`Reset()`).

A path condition and final symbolic expression is derived for each path through each method using symbolic execution. Each member variable is then partitioned as follows. First, all the path conditions are analyzed to look for usage of the variable. For each usage, intervals of the domain of the variable are formed such that the path condition is evaluated as true or false for all values in that interval. Finally, overlapping intervals are removed from their original partitions to form separate new intervals. The intervals for each member variable are conjoined with the intervals of every other member variable to form the overall set of states.

In this way, the member variable `curQtrs` is partitioned into two intervals, $[0, 0]$ and $[1, M]$, where M is the maximal possible value of `curQtrs`, through the condition on the `if` statement in the `AddQtr` method. Similarly `allowVend` is partitioned $[0, 0]$ and $[1, M]$, where M is the maximal possible value of `allowVend`, through the condition in the `Vend` method. The domain of the member variable `totalQtrs` forms one interval and is effectively ignored, since it does not appear in any conditions in the code. With two intervals for `curQtrs`, two intervals for `allowVend` and one interval for `totalQtrs`, four distinct states are formed.

Transitions are constructed by analyzing the path condition of each path, and finding the member variable intervals that satisfy that path condition in some state. This state becomes the pre-state of the transition. The post-state is then found by finding some state whose member variable intervals satisfy the final expression of the path. The derived object state diagram for the Coinbox example can be seen in Figure 4.2.

The main problem with the work of Kung *et al.* is the difficulty of analyzing dynamic behaviour, which is hard to achieve statically. Unbounded loops are dealt with in a limited manner through the consideration of a selected number of

```
class CCoinBox
{
  unsigned int totalQtrs; // total quarters collected
  unsigned int curQtrs;   // current quarters collected
  unsigned int allowVend; // 1 = vending is allowed

public:
  CCoinBox() {Reset();}
  void AddQtr();
  void ReturnQtrs() { curQtrs = 0; }
  unsigned int isAllowedVend() { return allowVend; }
  void Reset() { totalQtrs = 0; allowVend = 0; curQtrs = 0; }
  void Vend();
};

void CCoinBox::AddQtr()
{
  curQtrs = curQtrs + 1;
  if (curQtrs > 1)
    allowVend = 1;
}

void CCoinBox::Vend()
{
  if (isAllowedVend())
  {
    totalQtrs = totalQtrs + curQtrs;
    curQtrs = 0;
    allowVend = 0;
  }
}
```

Figure 4.1: The coinbox example

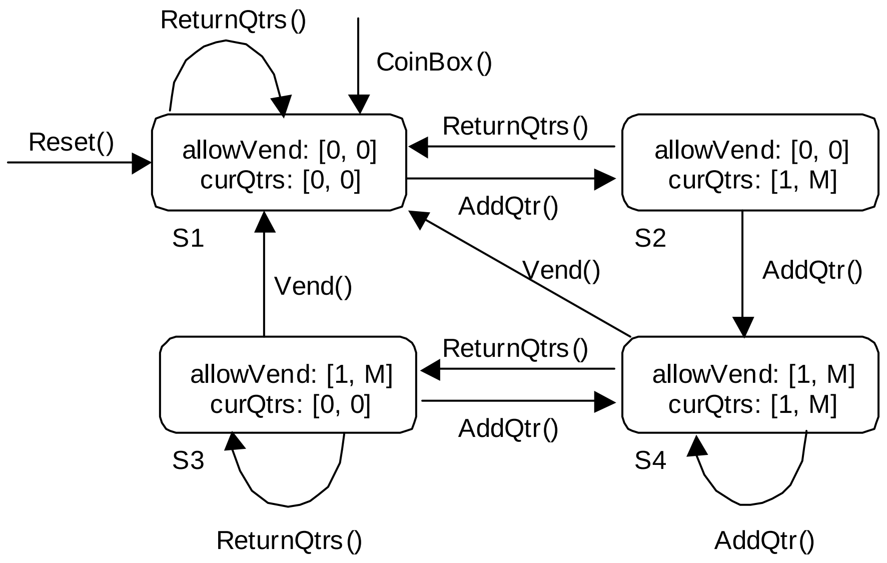


Figure 4.2: Object state diagram for the coinbox example of Figure 4.1

iterations, which inevitably results in an incomplete state model. Furthermore, the method only works with scalar input variables. A further problem is that the method only deals with internal state variables, whereas internal variables may also appear within function bodies, or be used as return values from other functions.

4.3.2 Reverse Engineering State Models for Model Checking

There is a body of work in the literature devoted to reverse engineering state models for model checking. Model checking is the process of exhaustively searching a finite state model of a system in order to check for violations of its requirements. Model checking is generally used to validate abstractions of hardware or software designs. However if one wants to model check actual programs, a model needs to be synthesized from the code.

Such derivations are presented with many complications, due to state explosion problems, and also the problem of the semantic gap that exists between the languages accepted by model checkers (for example Promela) which are generally static, and programming languages, which are highly dynamic.

One example of a program model checker is Bandera [CDH$^+$00], which model checks Java programs. Bandera attempts to build tractable models of Java code through processes of *component elimination*, *data abstraction* and *component restriction*. The initial stage of component elimination involves the removal of parts of the software in the final model that can have no effect on deciding whether the current property being checked will be violated or not. This is achieved through the use of program slicing [Wei84]. After these parts of the system have been removed, some remaining relevant variables may be abstracted to remove surplus information that is not required for the current property being checked. The user can define abstractions in a special language [DHJ$^+$01], and these are stored in a library for future use. Bandera then provides guidance on which abstractions to use for given programs and properties. The abstracted program is generated through the compilation of the abstract definitions into Java representations, and then by transforming concrete operations to new abstract ones. For example if an integer was abstracted to the values $\{neg, zero, pos\}$, the abstract *plus* operator would return *neg* for $neg + neg$, *plus* for $zero + pos$, a non-deterministic value for $neg + pos$ and so on. One danger here is over-abstraction to the point at which a property no longer becomes adequately checkable. If the first two stages still cannot produce a tractable model, a restricted version is used. This is derived through strategies such as limiting the ranges of variables, bounding execution steps or bounding the number of objects that can be created.

Similar tools to Bandera include Java PathFinder [HP00] and JCAT [DIS99], both of which translate concurrent Java code into the Promela model checking language. However these tools give the impression of not being as proficient in producing compact state models, concentrating mainly on control flow properties such as deadlock. FeaVer [HS01] is another model checking system, but gives the impression of being more suited to aiding model construction, as opposed to being a model constructor in its own right. Furthermore, Groce *et al.* [GPY02] propose a methodology called “Adaptive Model Checking”, which can learn from inconsistences between a system and its corresponding model in order to perform suitable updates to the model; using techniques from machine learning.

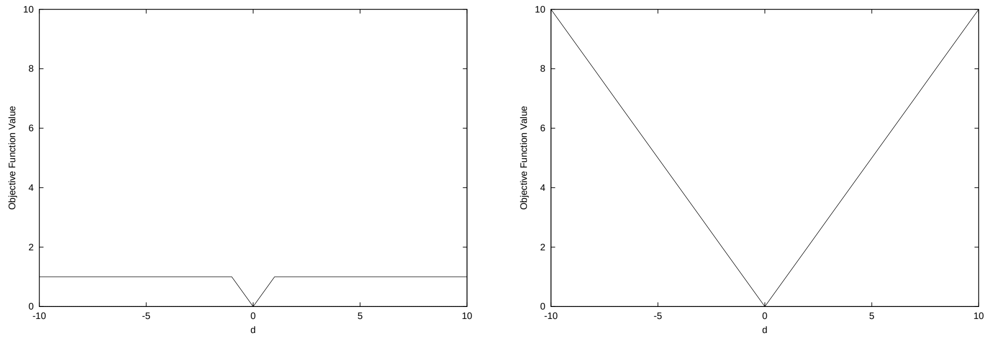
4.4 The Chaining Approach

The chaining approach [FK96a] was briefly introduced in Chapter 2, Section 2.3.3, and is a search-based structural test data generation method in its own right. It is of interest to this work because it incorporates a “backup” strategy when it encounters difficulties during the search for test data for a particular structure. These difficulties may possibly be due to the interference of internal variables, and the possible lack of guidance provided by the objective function.

The chaining approach uses local search and has only previously been used

```
CFG
Node
 s    void ca_initial_example(int x)
      {
 1        int flag = 0;

 2        if (x == 0)
 3            flag = 1;

 4        if (flag)
          {
 5            // target node
          }
      }
```

(a)



(b) (c)

Figure 4.3: Example for demonstrating the chaining approach. (a) Program code (b) Branch distance for the execution of node 4 as true (c) Branch distance for the execution of node 2 as true

to test functions with input-output behaviour. However, there is potential to extend the method by incorporating a global search in the form of evolutionary algorithms and testing the approach with state-based test objects.

The chaining approach was originally developed for Pascal programs. However for consistency all examples here are presented in C. The chaining approach is now explained in detail.

Consider the flag example of Figure 4.3a (this example has appeared already in this thesis, but is reproduced here for ease of reference). The target of the search is node 5. In order to execute node 5 the true branch from node 4 must be executed. However, the condition at node 4 involves a flag that is only true when the input value to the function is zero. Consequently the surface of the objective function derived from the branch distance at node 4 is flat since the flag is always false other than for the required value (Figure 4.3b). Therefore, the search receives no guidance and is likely to fail to find the required input value of `x`. Node 4 is referred to as a "problem" node. It can easily be seen that if node 3 is executed before 4, the true branch from node 4 will be executed. If the search tries to execute node 3, it needs to evaluate the condition at node 2 as true. The objective surface of the true branch predicate from node 2 is far more conducive to finding the desired value of `x` (Figure 4.3c), with a surface free of plateaux, descending down to the required input value. Therefore, if the search instead aims to execute node 3 before node 4, instead of merely node 4 alone, test data can be more easily found.

The basic idea of the chaining approach is to identify a sequence of nodes that can potentially change the outcome at problem nodes using a similar strategy as in the example above. By requiring that such a sequence of nodes is executed prior to the problem node, extra guidance can be made available to the search (for example via the true branch predicate from node 2 in the example), and the chances of finding test data for the target structure may be increased.

4.4.1 The Concept of a "Last Definition"

The process of node sequence identification is driven by data flow analysis. The set of nodes that can have an immediate effect on a problem node is the set of *last definition* nodes of variables used at the problem node. A "last definition" node i is a node that assigns a value to a variable v which may potentially be used by a node j. For the node to qualify as a last definition, a definition-clear path must exist between node i and node j with respect to v (the concept of a definition-clear path was defined in Section 2.3.1). In the example of Figure 4.3, only one variable is used at problem node 4 - the variable `flag`. Last definitions of `flag` from node 4 are nodes 1 and 3. Node 1 is definition-clear with respect to `flag` through to node 4 via node 2, avoiding node 3 (Figure 4.4). Node 3 is definition-clear since node 4 is its immediate predecessor.

In the example of Figure 4.5, nodes 6 and 8 are possible last definitions of the variable `flag` used at node 9. However nodes 1 and 3 cannot be last definitions since there is no definition-clear path from either node through to node 9 avoiding node 6, which redefines `flag`.

Sequences of last definition nodes are constructed using the notion of an event sequence.

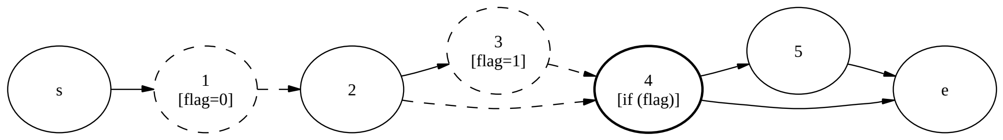


Figure 4.4: Finding last definitions. The problem node is node 4. The only variable used at the problem node is `flag`, last defined at node 1 (via the sub-path 1-2-4) and node 3.

```
CFG
Node
 s    void last_defs_example(int x)
      {
 1        int flag = 0;
 2        if (x == 0)
 3            flag = 1;

 4        if (flag)
          {
 5            // perform some action
          }

 6        flag = 0;
 7        if (x == 1)
 8            flag = 1;

 9        if (flag)
          {
10            // perform some action
          }

 e    }
```

Figure 4.5: Example for finding last definitions

4.4.2 Event Sequences

An event sequence can be thought of as a form of abstract path. An "event" simply refers to the execution of a node. Formally, an event sequence is a sequence of events, $< e_1, e_2, \cdots e_k >$ where each event is a tuple $e_i = (n_i, C_i)$ where n_i is a program node and C_i is a set of variables referred to as a constraint set [FK96a]. The constraint set is simply a set of variables that must not be modified until the next event in the sequence. That is to say, a definition-clear path must be executed between two events e_i and e_{i+1} with respect to each variable v in C_i.

The following event sequence $< (s, \emptyset), (1, \{flag\}), (4, \emptyset) >$ is an event sequence referring to nodes in the example of Figure 4.3. It requires that the start node s is executed, followed by the execution of node 1. Node 4 must then be executed - but avoiding any reassignment to `flag` before node 4. This means the false branch must be taken from node 2.

An event sequence is feasible if input data exists on which the event sequence can be successfully executed. The event sequence $< (s, \emptyset), (1, \{flag\}), (4, \emptyset) >$ is feasible, however the event sequence $< (s, \emptyset), (1, \{flag\}), (4, \emptyset), (5, \emptyset) >$ is not.

4.4.3 The Chaining Process

The chaining approach begins with an initial sequence E_0 which contains the start node s and the goal node. Both events have empty constraint sets. In the example of Figure 4.3, node 5 is the goal node. The initial event sequence is therefore:

$$E_0 \quad = \quad < (s, \emptyset), (5, \emptyset) >$$

If input data cannot be found to take the true branch from node 4 so that node 5 might be executed, node 4 is declared as a problem node. Node 4 is inserted into the event sequence:

$$< (s, \emptyset), (4, \emptyset), (5, \emptyset) >$$

Last definition nodes for node 4 are then identified - nodes 1 and 3. Two new event sequences are now generated, one demanding the execution of node 1 before node 4 (E_1), the other demanding the execution of node 3 before node 4 (E_2):

$$E_1 \quad = \quad < (s, \emptyset), (1, \{flag\}), (4, \emptyset), (5, \emptyset) >$$

$$E_2 \quad = \quad < (s, \emptyset), (3, \{flag\}), (4, \emptyset), (5, \emptyset) >$$

The inserted events are formed from the last definition node, and a constraint set formed from the variable defined at the node. The addition of the last definition variable into the constraint set specifies that it will not be modified again until the problem node is encountered; ensuring the effect of that last definition on the problem node is not destroyed.

In general, then, the event sequence begins an initial event sequence E_0 with just the start node s and the goal node g - $< (s, \emptyset), (g, \emptyset) >$. The test data search may fail to find inputs to execute the event sequence, with the flow of execution diverging down an unintended branch at node p_1. Node p_1 is declared as a problem node and is inserted into the event sequence - $< (s, \emptyset), (p_1, \emptyset), (g, \emptyset) >$. For the problem node p_1, the set of last definition nodes $lastdef(p_1)$ are found for the set of variables used at p_1. For each last definition $d_i \in lastdef(p_1)$, a new event sequence is generated containing an event associated with that last event:

$$\begin{aligned}
E_1 &= < (s, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (e, \emptyset) > \\
E_2 &= < (s, \emptyset), (d_2, \{def(d_2)\}), (p_1, \emptyset), (e, \emptyset) > \\
\ldots & \\
E_N &= < (s, \emptyset), (d_N, \{def(d_N)\}), (p_1, \emptyset), (e, \emptyset) >
\end{aligned}$$

Assuming that each definition modifies only one variable, the constraint set associated with each last definition d_i in E_i is a one element set $def(d_i)$ that requires a variable defined by d_i is not modified between d_i and p_1.

The chaining approach selects one of the event sequences and tries to find inputs for which it is successfully executed. If such an input is found, then test data to execute the test goal has been found. If not, new event sequences may be generated. For example, in trying to find inputs to execute E_1, a new problem node p_{1_1} may be encountered before d_1 can be executed. If this is the case p_{1_1} is inserted into the sequence:

$$< (s, \emptyset), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\}), (p_1, \emptyset), (e, \emptyset) >$$

Last definitions of variables are then found for p_{1_1}, and new events are generated and inserted into a new set of event sequences:

$$E_{1_1} \quad = \quad < (s, \emptyset), (d_{1_1}, \{def(d_{1_1})\}), (p_{1_1}, \emptyset), (d_1, \{def(d_1)\})(p_1, \emptyset), (e, \emptyset) >$$

$$\begin{aligned}
E_{1_2} &= <(s,\emptyset),(d_{1_2},\{def(d_{1_2})\}),(p_{1_1},\emptyset),(d_1,\{def(d_1)\})(p_1,\emptyset),(e,\emptyset)> \\
\ldots \\
E_{1_N} &= <(s,\emptyset),(d_{1_N},\{def(d_{1_N})\}),(p_{1_1},\emptyset),(d_1,\{def(d_1)\})(p_1,\emptyset),(e,\emptyset)>
\end{aligned}$$

Problem nodes can arise as a result of a failure to execute an event node, or as a result of a constraint set violation. The example of Figure 4.6 could lead to a constraint set violation. Suppose the current event sequence is:

$$<(s,\emptyset),(4,\{flag1\}),(9,\emptyset),(10,\emptyset)>$$

The constraint set for the event corresponding to node 4 requires that the variable `flag1` is not modified from node 4 up until node 9. However test data cannot be found to avoid the re-definition of `flag1` at node 8. This is because the condition at node 7 is also dependent on a flag variable - `flag2` - which has a low probability of being false. This is due to the likely situation where the input variable `b` is not zero, and `flag2` is set to true at node 6. Node 7 is therefore declared as a problem node, for which the further event sequences are generated, namely:

$$\begin{aligned}
E_x &= <(s,\emptyset),(2,\{flag2\}),(4,\{flag1,flag2\}),(7,\{flag1\}),(9,\emptyset),(10,\emptyset)> \\
E_y &= <(s,\emptyset),(4,\{flag1\}),(6,\{flag1,flag2\}),(7,\{flag1\}),(9,\emptyset),(10,\emptyset)>
\end{aligned}$$

event sequence E_x being feasible and E_y infeasible.

Generated event sequences are organized in a tree structure (Figure 4.7). At the first level are the event sequences generated as a result of the first problem node, with subsequent levels formed if further problem nodes are encountered. Event sequences are explored in the tree in a depth-first fashion to a maximum depth limit.

A flow chart of the overall process of the chaining approach can be seen in Figure 4.8.

4.4.4 Formal Generation of an Event Sequence

The general strategy for generating event sequence can be formally described as follows. Let $E =< e_1, e_2, ..., e_{i-1}, e_i, e_{i+1}, ..., e_m >$ be an event sequence. Suppose the test data search finds input data to partially execute the event sequence up to event e_i, with a problem node p encountered between events e_i and e_{i+1}. Let d be a last definition of problem node p. A new event sequence is

```
CFG
Node
 s     void constraint_set_violation_example(int a, int b)
       {
 1         int flag1 = 0;
 2         int flag2 = 0;

 3         if (a == 0)
 4             flag1 = 1;

 5         if (b != 0)
 6             flag2 = 1;

 7         if (flag2)
 8             flag1 = 0;

 9         if (flag1)
10             // target statement
 e     }
```

Figure 4.6: Constraint set violation example

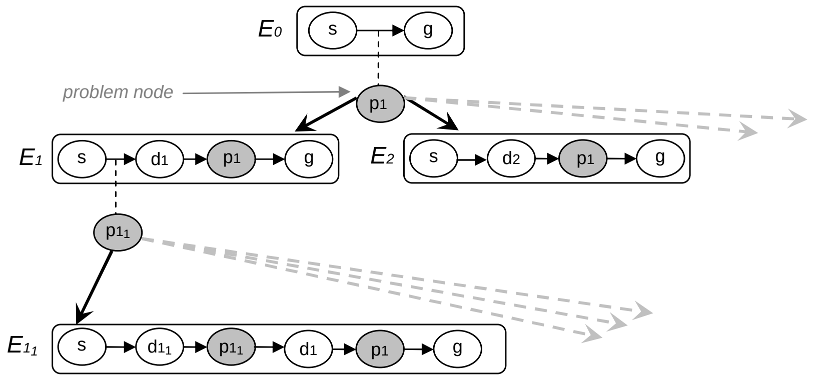


Figure 4.7: The chaining tree (adapted from Reference [FK96a])

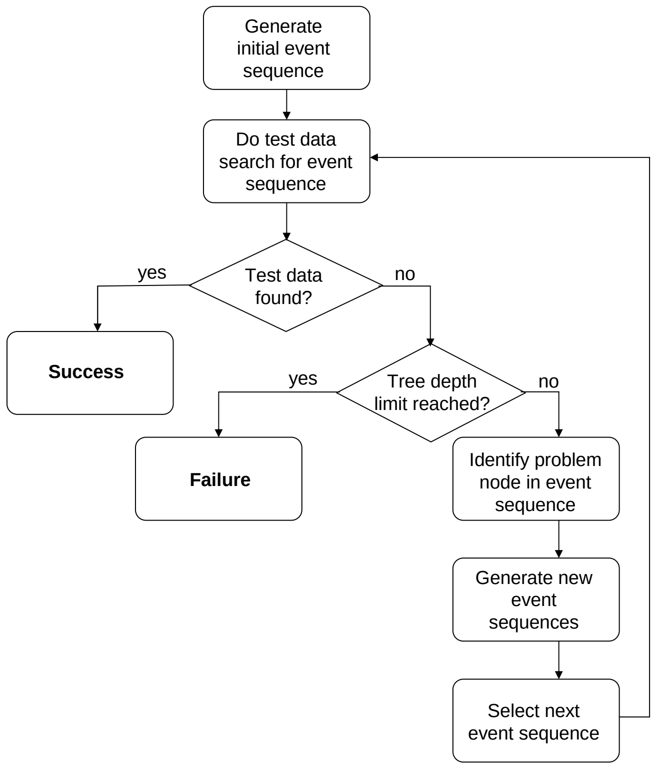

Generate initial event sequence
Do test data search for event sequence
Test data found?
yes
no
Success
Tree depth limit reached?
yes
no
Failure
Identify problem node in event sequence
Generate new event sequences
Select next event sequence

Figure 4.8: The chaining process

generated from E by inserting two events into sequence: $e_d = (d, def(d))$ and $e_p = (p, \emptyset)$. Event e_p is always inserted between events e_i and e_{i+1}. In general, however, event e_d may be inserted in any position between e_1 and e_{k+1} in the event sequence. Therefore, the following event sequence is generated:

$$E' =< e_1, e_2, ..., e_{k-1}, e_k, e_d, e_{k+1}, ..., e_{i-1}, e_i, e_p, e_{i+1}, ..., e_m >$$

Insertion of new events into the sequence requires modification of certain constraint sets as part of events already in the sequence.

The constraint set of the event corresponding to the problem node is simply the same as the constraint set of the prior event in the sequence:

$$C_p = C_i \tag{4.1}$$

Previously, variables in C_i could not be modified between e_i and e_{i+1}. The above step ensures this is still the case, by maintaining that these variables will not be modified between e_p and e_{i+1} either.

The constraint set C_d for the event e_d is formed from the variable defined at the event node, $def(d)$, merged with the variables of the constraint set of the previous event, C_k:

$$C_d = C_k \cup def(d) \tag{4.2}$$

In a similar fashion to Equation 4.1, this rule maintains consistency of the event sequence by ensuring that variables in C_k are not modified between e_k and the new event e_d. However, the variable $def(d)$ might still be modified between e_{k+1} and e_p, ruining the effect of the last definition. The final step prevents this by adding $def(d)$ to each constraint set for each event from e_{k+1} up to but not including e_p:

$$\forall j, k+1 \leq j \leq i, C_j \cup def(d) \tag{4.3}$$

Consider the example of Figure 4.9. Suppose the target is to execute node 11. Initially node 7 was a problem node and was inserted into the event sequence. The current event sequence is as follows, with the last definition of `flag1` at node 4 also inserted into the sequence:

$$< (s, \emptyset), (4, \{flag1\}), (7, \emptyset), (11, \emptyset) >$$

Execution may now proceed through the true branch from node 7, but then

```
CFG
Node
 s    void multiple_problem_nodes_example(int x, int y, int z)
      {
 1        int flag1 = 0;
 2        int flag2 = 0;

 3        if (x == 0)
 4            flag1 = 1;

 5        if (y == 0)
 6            flag2 = 1;

 7        if (flag1)
          {
 8            if (z != 0)
 9                flag2 = 0;

10            if (flag2)
              {
11                // target node
              }
          }
 e    }
```

Figure 4.9: Multiple problem node example

node 10 might be problematic. If this is the case, node 10 is inserted into the sequence between nodes 7 and 11. According to Equation 4.1 the constraint set of the new event is the same as the prior event. The constraint set of node 7 is empty, and so the intermediate event sequence is:

$$< (s, \emptyset), (4, \{flag1\}), (7, \emptyset), (10, \emptyset), (11, \emptyset) >$$

The variable `flag2` is used at problematic node 10. This is last defined at nodes 2 and 6. Suppose node 6 is to be inserted into the sequence. The only place it can go in the sequence is between the events corresponding to nodes 4 and 7. In accordance with Equation 4.2, the constraint set of the new event is that of the prior event, $\{flag1\}$ coupled with the defined variable - $flag2$. The constraint set of node 7 must be updated according to Equation 4.3 with the variable defined at the earlier event (note that this means the improbable false branch should be taken from node 8).

The newly generated event sequence is therefore:

$$< (s, \emptyset), (4, \{flag1\}), (6, \{flag1, flag2\}), (7, \{flag2\}), (10, \emptyset), (11, \emptyset) >$$

4.4.5 Repetition of a Problem Node

The chaining approach assumes a problem node can only occur in an event sequence once. In practice this prevents the generation of some nonsensical event sequences. Take the example of Figure 4.10. Suppose the target of the search is node 7. Node 6 becomes a problem node, for which further event sequences can be generated. One of these new event sequences is:

$$< (s, \emptyset), (10, \{flag\}, (6, \emptyset), (7, \emptyset) >$$

since node 9 might be reached in the previous iteration of the loop. However node 9 is nested within the `if` statement from node 6, which will be found to be a problem node again when the search attempts to find input data. Therefore, no further event sequences are generated from:

$$< (s, \emptyset), (6, \emptyset), (10, \{flag\}, (6, \emptyset), (7, \emptyset) >$$

4.4.6 Test Data Search

The chaining approach uses a local search method known as the alternating variable method to find test data. The alternating variable method was intro-

```
CFG
Node
 s    void problem_node_repetition_example(int a[10])
      {
 1        int i;
 2        int flag = 0;

 3        for (i=0; i < 10; i++)
          {
 4            if (flag)
              {
 4                if (a[i] == 0)
 5                    flag = 1;

 6                if (flag)
                  {
 7                    flag = 0;

 8                    // perform some action
                      // ...

 9                    if (i < 10 && a[i+1] == 0)
10                        flag = 0;
                  }
              }
          }
 e    }
```

Figure 4.10: Problem node repetition example

duced in Section 2.3.3.

An initial input vector is chosen at random. The alternating variable method then attempts to find input data so that each event in the sequence is executed in turn through the program. In order for this to happen, each branch is classified with respect to each pair of adjacent events $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$ in the sequence as either critical, semi-critical or non-essential. This branch classification assumes n_i has been successfully executed, and the goal is to executed n_{i+1} without modifying any of the variables in C_i

Critical Branches

A branch (p, q) is declared critical with respect to e_i and e_{i+1} if and only if:

- There does not exist a definition-clear path from p to n_{i+1} or n_i with respect to C_i through branch (p, q); and
- There exists a definition-clear path from p to n_{i+1} with respect to C_i from the alternative branch of p

If a critical branch is executed, the alternating variable local search method is invoked to modify the input vector so that the alternative branch is taken. In the example of Figure 4.3 and the event sequence:

$$< (s, \emptyset), (3, \{flag\}), (4, \emptyset), (5, \emptyset) >$$

the false branch from node 2 is critical for the execution of node 3 of the second event e_2. The false branch from node 2 results in a path that will never reach node 3. If the initial input is `(x=10)`, the branch distance is 10 (refer to Table 2.1 for Korel's objective functions for branching predicates). The alternating variable method makes exploratory moves around this value. A lower value of `x` leads to an improvement, and decreasing values of `x` are sought until the value of `0` is reached.

In the example of Figure 4.6 and the event sequence:

$$< (s, \emptyset), (4, \{flag1\}), (9, \emptyset), (10, \emptyset) >$$

the true branch from node 7 is critical since it leads to a redefinition of the variable `flag1`.

If the alternating variable cannot find an input vector so that a critical branch is not taken, node p is declared as problematic, and further event sequences at the next level of the chaining tree generated.

Semi-Critical Branches

A branch (p, q) is defined as *semi-critical* with respect to e_i and e_{i+1} if it is not critical, and:

1. n_{i+1} is control dependent on p; and
2. There does not exist an acyclic definition-clear path from p to n_{i+1} with respect to C_i through (p, q).

When a semi-critical branch is executed, the alternating variable method is invoked to adapt the input vector so that the alternative branch might be taken. However, if no such input can be found, node p is not declared as problem node - execution continues in the hope of simply taking the alternative branch in the subsequent iteration of the loop. Clearly, however, if the loop has to iterate, the chances of encountering a critical branch are increased, and so it is preferred that semi-critical branches are avoided.

Non-Essential Branches

A *non-essential* branch is simply any branch which is not critical or semi-critical. If a non-essential branch is taken there is no need to adjust the input vector since the branch is not important with respect to the current event.

4.4.7 Test Data Generation for Covering Branches and Definition-Use Pairs

The chaining approach has been presented for finding test data for specific nodes within programs, as part of a statement coverage strategy. The method can also deal with other structural targets such as branches or definition-use pairs.

In order to perform branch coverage, branches need to be inserted as test goals into event sequences. This does not require an extension of the event and event sequence model. A branch is inserted into an event sequence by simply inserting a pair of events $e_i = (n_i, \emptyset)$ and $e_j = (n_j, \emptyset)$ into the sequence, where n_i and n_j are the respective originating and terminating nodes of the branch. The initial event sequence is therefore:

$$< (s, \emptyset), (n_i, \emptyset), (n_j, \emptyset) >$$

No events will be inserted into the sequence between e_i and e_j, which would destroy the intention to execute the edge between nodes n_i and n_j. This is because e_j will either be executed directly after e_i, or, e_i will have been declared a problem node, in which case new events will be inserted before it.

In order to perform definition-use coverage, a definition-use pair (d, u) is inserted:

$$< (s, \emptyset), (d, def(d)), (u, \emptyset) >$$

4.4.8 Evaluation of the Chaining Approach

Ferguson and Korel [FK96a] found their chaining approach achieved a higher level of coverage for a series of examples when compared to random test data generation, the constraint-based testing approach (described in Section 2.3.2), and the goal-oriented approach (described in Section 2.3.3) on which it was originally based.

The chaining approach could be extended to deal with test objects with states. It has advantages over the other techniques considered in this chapter because:

- It adapts to the outcome of a search, as opposed to trying to guess the difficulties that might be encountered beforehand. This makes the approach applicable to a wider range of internal variable problems other than those involving just flags;
- No information needs to be reverse engineered from the program code, other than knowledge of the data dependencies within it.

Korel [Kor96] shows how the last definition analysis works for programs with embedded procedure calls, with the analysis tracing the data flows into and back out of the called procedure. Ferguson and Korel [FK96b] also show how the method can be extended to handle distributed programs.

4.5 Conclusions

This chapter has evaluated several techniques to help overcome the internal variable problem for evolutionary testing. Several solutions for dealing with the flag variant of the problem were considered. However these techniques rely on knowing that the flag variable will induce a flat objective function landscape. For more general forms of internal variable it is uncertain what the form of the objective function landscape take, and whether it will indeed provide insufficient guidance to the search.

Several techniques were considered which reverse engineer state models from code. Such analysis may provide valuable internal information regarding how to put the test object in certain states and overcome internal variable problems. However these techniques suffer from state explosion problems, and the difficulty of analyzing dynamic features of code.

Finally, the chaining approach was considered. The chaining approach is a search-based structural test data generation method itself, but utilizes local search. It is of interest to this work because it incorporates a “backup” strategy when it encounters difficulties during the search for test data for a particular structure, difficulties that are possibly due to the existence of internal variables and a lack of guidance provided by the objective function. The basic idea of the chaining approach is to find a sequence of statements involving internal variables that need to be executed prior to the test goal. By requiring that these statements are executed, information previously unavailable to the search can be utilized, possibly guiding it into potentially promising and unexplored areas of the test object’s input domain. The chaining approach could be extended to deal with test objects with states. It adapts to the outcome of a search, as opposed to trying to guess the difficulties that might be encountered beforehand. This makes the approach applicable to a wider range of internal variable problems other than those involving just flags. No information needs to be reverse engineered from the program code, other than knowledge of the data dependencies within it.

Chapter 5

Hybridizing Evolutionary Test Data Generation with an Extended Chaining Approach

5.1 Introduction

The last chapter proposed the incorporation of a chaining method within evolutionary structural test data generation, in order to overcome internal variable problems. This chapter presents this approach. The original local search used by the chaining approach is replaced in favour of an evolutionary search. An evolutionary search needs to be able to make global comparisons with respect to test inputs and event sequences, requiring the definition of a new, more sophisticated objective function. Weaknesses and limitations of the original chaining algorithm are identified, and addressed in an extended chaining algorithm. The goal is to overcome internal variable problems for test objects with states. However, as proof of concept the initial approach is tested with nine test objects with input-output behaviour. Due to the presence of internal variables, each test object contains a statement that is problematic for the standard evolutionary approach to cover.

5.2 The Hybrid Approach

This section introduces the proposed "hybrid approach".

5.2.1 Test Data Search

The original local search of the chaining approach is replaced in favour of an evolutionary search. As with the original approach, a new test data search takes place for each new event sequence encountered. The strategy taken by the original local approach was to adapt a single test input vector to execute the event sequence, by minimizing critical and semi-critical branch distances one after the other. However, an evolutionary search needs to be able to make global comparisons with respect to test inputs and event sequences, requiring a more sophisticated objective function. The objective value of an input vector $\mathbf{x}$ for an event sequence E of length l is computed using the minimizing objective function formula:

$$\sum_{i=1}^{l} obj\_event(e_i) \tag{5.1}$$

where e_i is the ith event in the event sequence, and $obj\_event(e)$ is calculated for $e_i = (n_i, C_i)$ as follows:

1. If the event node n_i - to be executed after the event node of e_{i-1} but before e_{i+1} - is missed, add the result of the node-oriented objective function of Wegener *et al.* [WBS01], i.e.

   $$approach\_level + normalize\_bd(branch\_dist)$$

   where $approach\_level$ is the approach level for node n_i, and $branch\_dist$ is the branch distance of the alternative branch at which execution diverged away from n_i. This function was originally documented in Section 2.3.5. The approach level is calculated using Equation 2.2, and the $normalize\_bd$ function is defined in Equation 2.1.

2. For each definition node d executed for each variable $v \in C_i$ violating the definition-clear path required until e_{i+1}, add the normalized branch distance for the alternative branch at the last branching node that led to d's execution.

Note that the objective function for the initial event sequence $<(s, \emptyset), (g, \emptyset)>$ for some target node g is equivalent to the node-oriented objective function for g for the standard evolutionary approach; since s is always executed and none of the events have constraint sets.

Example

Take the example of Figure 5.1 and the event sequence:

$$< (s, \emptyset), (3, \{flag\}), (6, \emptyset), (7, \emptyset) >$$

Take the input `(i=10, j=20)`. The first event, e_1 is the start node and is always executed. However for e_2, node 3 is missed, with the false branch from node 2 taken. The approach level is zero and the branch distance is $10 + K$ (using Tracey's branch distance calculations of Table 2.2). Furthermore, the constraint set of e_2 is violated, since node 5 is executed, which redefines the value of `flag`. Therefore, the normalized branch distance of the alternative false branch from node 4 is added - the branch distance being $20 + K$. Node 6 of event e_3 is successfully reached, but node 7 of event e_4 of is missed, since the false branch from node 6 is taken. The approach level is 1 and the branch distance is $1 + K$. Therefore:

$$\begin{aligned}
obj\_event(e_1) &= 0 \\
obj\_event(e_2) &= normalize\_bd(10 + K) + normalize\_bd(20 + K) \\
obj\_event(e_3) &= 0 \\
+obj\_event(e_4) &= 1 + normalize\_bd(1 + K) \\
\hline
&= 1.0377 \ (where\ K = 1)
\end{aligned}$$

Figure 5.2 shows that as the value of `i` and `j` tend to zero, the objective value of e_2 also tends to zero, until finally e_4 is also successfully executed and input data has been found for the entire event sequence.

5.2.2 Identification of Problem Nodes and Variables Used at the Problem Node

The evolutionary search works to minimize the objective function. If an objective value of zero can be found, test data to execute the test goal will also have been found. If this is not the case further event sequences are generated using the first problem node encountered by the best individual found during the search.

The test data corresponding to the best individual may result in an ex-

```
CFG
Node
 s    void objective_function_example(int i, int j)
      {
 1        int flag = 0;

 2        if (i == 0)
 3            flag = 1;

 4        if (j > 0)
 5            flag = 0;

 6        if (flag)
          {
 7            if (j == 0)
 8                // target statement
          }
 e    }
```

Figure 5.1: Example program for calculating the objective value of an event sequence

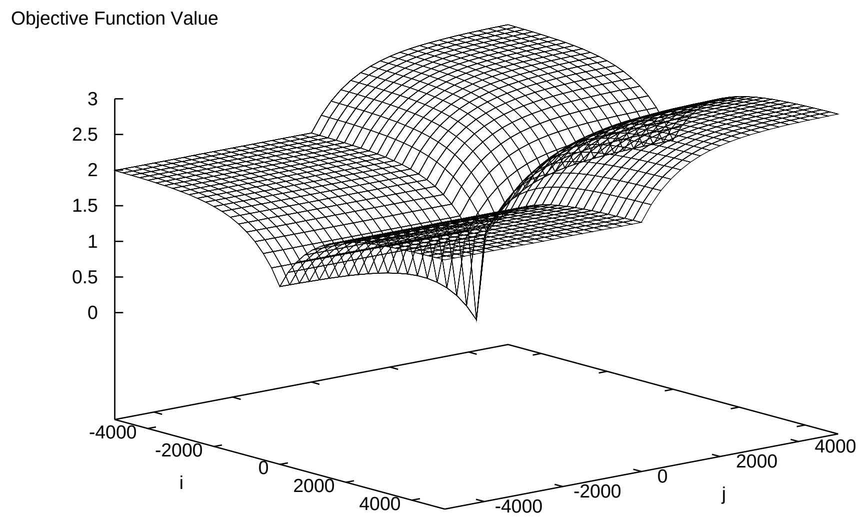


Figure 5.2: Objective function landscape for the example program of Figure 5.1 and the event sequence $< (s, \emptyset), (3, \{flag\}, (6, \emptyset), (7, \emptyset) >$

ecution path which diverges away from the intended path at several points - resulting in several potential problem nodes. However, as for the original chaining approach, only the first problem node is of importance, with definitions of variables used at this problem node used as the basis for the generation of further event sequences.

5.2.3 Extensions to the Original Chaining Approach

In this section, some shortcomings of Ferguson and Korel's original chaining algorithm [FK96a, Kor96, FK96b] are highlighted. Extensions are proposed to deal with these weaknesses.

Handling Logical AND and Logical OR operators

The papers describing the original chaining approach [FK96a, Kor96, FK96b] did not describe a mechanism for handling conditions composed using logical AND or logical OR operators. The hybrid approach allows the use of such conditions and extends the behaviour of the original approach in these instances. When a problem node is identified, last definitions are only sought for the variables used in the evaluated, unsatisfied sub-conditions of the overall condition.

Take the example of Figure 5.3. Suppose the best input found by the evolutionary algorithm for the event sequence:

$$< (s, \emptyset), (6, \{flag2\}), (9, \emptyset), (10, \emptyset) >$$

is `(a=100, b=1, c=1)`. This input vector diverges away from the intended path set out by the event sequence down the false branch from node 5, leading to a miss of node 6; down the true branch of node 7, leading to an unintended hit of 8 - the re-definition of the variable `flag2`; and finally down the false branch of node 9, leading to a miss of the target node 10. Therefore there are three problem nodes, node 5, 7 and 9. However only the first problem node - node 5 - is used for the generation of further event sequences.

The variables used at problem node 5 include `flag1` and `b`. However the sub-condition using `b` was not evaluated since the short-circuit `&&` operator broke off when it found the initial condition where the `flag1` needs to be true was false. Therefore last definitions are only sought for `flag1`. If the condition had been composed in the reverse manner, i.e.:

```
    if (b == 1 && flag1)
```

```
CFG
Node
 s    void problem_node_example(int a, int b, int c)
      {
 1        int flag1 = 0;
 2        int flag2 = 0;

 3        if (a == 1)
 4            flag1 = 1;

 5        if (flag1 && b == 1)
 6            flag2 = 1;

 7        if (c != 0)
 8            flag2 = 0;

 9        if (flag2)
10            // target statement
 e    }
```

Figure 5.3: Example code for finding problem nodes

then `flag1` would still be the only variable subject to the search for last definitions, since condition `b == 1` is satisfied by the input vector. Last definitions of `flag1` occur at nodes 1 and 4, leading to the generation of two new event sequences, namely:

$$< (s,\emptyset), (1,\{flag1\}), (5,\emptyset), (6,\{flag2\}), (9,\emptyset), (10,\emptyset) >$$

and

$$< (s,\emptyset), (4,\{flag1\}), (5,\emptyset), (6,\{flag2\}), (9,\emptyset), (10,\emptyset) >$$

The method for computing branch distances in the presence of logical connectives is described in Section A.3.3.

Returning to a Problem Node

In Korel's original algorithm a problem node cannot appear in an event sequence more than once. Furthermore, the original approach declares failure when a problem node remains problematic for a feasible event sequence. Take the example of Figure 5.5. The target is node 7, which is only executed when half or more of the inputted integer array values are zero. This makes node 6 a problem node for which further event sequences are generated, these being:

$$
\begin{aligned}
E_1 &= < (s, \emptyset), (1, \{counter\}), (6, \emptyset), (7\emptyset) > \\
E_2 &= < (s, \emptyset), (5, \{counter\}), (6, \emptyset), (7\emptyset) >
\end{aligned}
$$

E_1 is infeasible. E_2 is not infeasible, but requires that the counter variable is incremented at only once. This is unlikely to be enough to ensure that input data can be found - the chances of four more array values being zero in a large input domain being extremely slim. Node 6, however, is still a problem node. The original algorithm declares failure at this point since a problem node cannot be revisited.

In the extended approach, a problem node can be revisited. Further event sequences can be generated from E_2. However another difficulty is encountered. A further increment `counter` cannot be inserted between the events corresponding to nodes 5 and 6, because `counter` appears in the constraint set for the event corresponding to node 5. Furthermore, a further increment could not appear between nodes s and 5, as the definition is not a last definition, since the last definition of `counter` already appears in the sequence at node 5. To handle this problem another extension is made to the algorithm, using the notion of *influencing sets*.

Extended Event Sequence Generation using Influencing Sets

The original chaining approach inserts new events that correspond to program nodes that are last definitions for variables used at the problem node. However there is potentially a greater set of variables that can affect the outcome at the problem node. For example the values of `a` and `b` influence the value of `x` and thus have an intermediate effect on the outcome of the following `if` statement:

```
x = a + b;

if (x > 0) {
  // ...
}
```

Given a program node and some path to the problem node, an influencing set consists of *all* variables that can affect the outcome at the problem node. The event sequence generation process is forced to consider definitions for all variables that can potentially affect the problem node, allowing event sequences to be generated that were not possible with the original approach.

The recursive algorithm for the extended event sequence generation approach can be seen in Figure 5.4. Paths are explored backwards from the problem node. The influencing set is adapted according to the path taken.

For a newly identified problem node, the influencing set is simply the set of variables involved in evaluated, unsatisfied conditions at the problem node. Beginning with the current problem node sn, the initial influencing set I, and the event prior to the problem node event in the event sequence $e = (n, C)$, the algorithm traces its way in a backwards manner through the nodes of the program. The set *prev_nodes* is simply the set of program nodes connected to the current node by an outgoing edge. Each node pn in *prev_nodes* is analyzed.

Firstly, the algorithm checks to see if the pn is the same as the prior event node n. If this is the case, and the variable defined at n ($def(n)$) - is contained in the influencing set, the influencing set is modified by removing $def(n)$ and adding the uses of n ($uses(n)$). This is because no prior definition of $def(n)$ can now affect the outcome at the problem node, since n is itself a last definition. However, the variables used at n can affect the problem node because they are used in the assignment to $def(n)$. The original chaining algorithm does not consider last definitions of variables used at the set of last definition nodes originally identified. The procedure then recurses using the event node pn as the current node sn, the new influencing set and the new prior event in the sequence.

If pn is not the prior event node, the algorithm checks the constraint set of the preceding event in the event sequence. If pn defines any variables in the constraint set, this particular line of enquiry terminates, since the path to the next event is not definition-clear.

If pn does not define any variable in the constraint set, but instead defines a variable in the influencing set, a qualifying definition node has been found, a new event sequence can be generated using pn if pn is reachable from n (the event node of e) via some definition clear path with respect to C. This new event sequence is generated following the standard rules of the original approach (Equations 4.1 - 4.3) documented in the last chapter.

If none of the above cases are true, the procedure recurses using the new node pn as the current node sn, along with the unmodified values of I and e.

Finally, a global data structure of "search points" ensures that only acyclic program paths are considered between adjacent events in the event sequence, and that the algorithm terminates.

It is now demonstrated how influencing sets and the extended event sequence

Let E be the original event sequence from which new event sequences are required

Let S be a global set of search points, where a search point is a tuple $sp = (sn, I, e)$, where sn is a program node, I is an influencing set of variables, and $e = (n, C)$ is an event in the original event sequence E

```
Procedure $generate\_event\_sequences($
                    In: a search point, $sp = (sn, I, e = (n, C)))$
    Let $prev\_nodes$ be the set of control flow graph nodes connected to $sn$
    by an outgoing edge
    If $sp \notin S$
        $S \leftarrow S \cup sp$
        Repeat
            Let $pn$ be a program node, $pn \in prev\_nodes$
            $prev\_nodes \leftarrow prev\_nodes - \{pn\}$
            If $pn = n$
                If $def(pn) \in I$
                    $I \leftarrow I - \{def(pn)\}$
                    $I \leftarrow I \cup uses(pn)$
                End If
                $generate\_event\_sequences((pn, I, prev\_event(E, e)))$
            Else If $\forall v \in C, v \neq def(pn)$
                If $\exists v \in I, v = def(pn)$
                    If $reachable(pn, e)$
                        $create\_new\_event\_sequence(pn, E, e)$
                    End If
                    $generate\_event\_sequences((pn, I - \{def(pn)\}, e))$
                Else
                    $generate\_event\_sequences((pn, I, e))$
                End If
            End If
        Until $prev\_nodes = \emptyset$
    End If
End Procedure
```

$def(n)$ returns the variable defined at program node n (or $\emptyset$ if one is not defined), and $uses(n)$ returns the set of variables used by a program node n.

$reachable(pn, e)$ checks if a node pn can be reached from another node n of an event $e = (n, C)$ without violation of the constraint set C.

$create\_new\_event\_sequence(pn, E, e)$ creates a new event sequence for the next level of the tree from a definition node pn, the original event sequence E and the event e after which the new event should be inserted.

$prev\_event(E, e)$ returns the event prior to the event e in an event sequence E.

Figure 5.4: Recursive procedure for generating event sequences using influencing sets

generation procedure has practical benefit.

The last section described how further event sequences could be derived from the event sequence:

$$E_2 =< (s, \emptyset), (5, \{counter\}), (6, \emptyset), (7\emptyset) >$$

in the counter example. The problem node is node 6, and the influencing set is $\{counter\}$. In searching for program nodes back from node 6, node 5 is encountered. An event for node 5 appears in the event sequence. The definition variable is removed from the influencing set:

$$\begin{aligned} I &\leftarrow I - def(5) \\ &\leftarrow \{counter\} - \{counter\} \\ &\leftarrow \emptyset \end{aligned}$$

leaving an empty set. Uses at node 5 are added:

$$\begin{aligned} I &\leftarrow I \cup uses(5) \\ &\leftarrow \emptyset \cup \{counter\} \\ &\leftarrow \{counter\} \end{aligned}$$

Last definitions can be sought for variables in the influencing set from node 5. These last definitions include node 1, and node 5 on the previous iteration of the loop:

$$\begin{aligned} E_{2_1} &= < (s, \emptyset), (1, \{counter\}), (5, \{counter\}), (6, \emptyset), (7\emptyset) > \\ E_{2_2} &= < (s, \emptyset), (5, \{counter\}), (5, \{counter\}), (6, \emptyset), (7\emptyset) > \end{aligned}$$

In this example the influencing set is effectively unchanged - the variable `counter` is defined and used in the same statement, and so is removed and then re-added. However, consider the example of Figure 5.12. The target is node 10. Node 9 is a problem node. The generated event sequences are:

$$\begin{aligned} E_1 &= < (s, \emptyset), (3, \{shutdown\}), (9, \emptyset), (10, \emptyset) > \\ E_2 &= < (s, \emptyset), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) > \end{aligned}$$

E_1 is infeasible. E_2 is feasible, but assume node 9 remains problematic. In event sequence generation, node 8 is encountered tracing backwards from node 9. The influencing set is modified to remove the definition of *shutdown* and add the uses - $\{error1, error2\}$. This means that further event sequences can be generated:

$$\begin{aligned} E_{2_1} &= < (s,\emptyset),(5,\{error1\}),(8,\{shutdown\}),(9,\emptyset),(10,\emptyset) > \\ E_{2_2} &= < (s,\emptyset),(7,\{error2\}),(8,\{shutdown\}),(9,\emptyset),(10,\emptyset) > \end{aligned}$$

Assuming node 9 remains problematic, the following event sequence will be generated from both E_{2_1} and E_{2_2}:

$$< (s,\emptyset),(5,\{error1\}),(7,\{error1,error2\}),(8,\{shutdown\}),(9,\emptyset),(10,\emptyset) >$$

which explicitly requires nodes 5 and 7 to be executed before node 8, which in turn assures that the true branch is taken from node 9, and that node 10 will eventually be executed.

5.3 Experimental Study 3 - Test Data Search using the Hybrid Approach

Experiments were performed comparing the hybrid approach with the standard evolutionary approach.

A set of nine synthetic test objects were used with input-output behaviour. The goal of the experiments is to find test data to execute a specific statement within each program.

The statement in each test object is designed to be difficult for the standard evolutionary approach to cover, due to the presence of objective function landscapes that are deceptive, coarse or flat. These landscapes are caused by the use of internal variables such as counters, boolean flags or enumerations.

This is the same landscape initially encountered by the hybrid approach, because the objective function of the initial event sequence is equivalent to the node-oriented objective function used by the standard evolutionary approach. However, as has been described, the hybrid approach can create further event sequences that require certain statements to be executed which can potentially affect the outcome of the test goal - for example a particular assignment to a

```
CFG
Node
 s    void counter(int a[10])
      {
 1        int counter = 0;
 2        int i;

 3        for (i = 0; i < 10; i++)
          {
 4            if (a[i] == 0)
 5                counter ++;
          }

 6        if (counter == 5)
 7            // target statement
 e    }
```

Figure 5.5: Program code for the “Counter” test object

flag variable. The objective function attempts to guide the search to the prior execution of these statements, thus changing the form of the landscape and potentially improving the chances of finding test data.

5.3.1 Test Objects

Counter

The “Counter” test object (Figure 5.5) was introduced in Section 5.2.2. The target is node 7, which is dependent on a counter variable being above a certain threshold. However, the counter variable is only incremented under special circumstances - when values of the inputted integer array are equal to zero. Therefore, the objective landscape for the standard evolutionary approach - and the initial event sequence - is extremely coarse, consisting of plateaux corresponding to the different values of the counter. The landscape is plotted in Figure 5.16a for two dimensions corresponding to two of the array values, depicting two of the plateaux.

With the input being an array of ten integers in the range -15,000 to 15,000, the search space size is approximately 6×10^{44}.

Deceptive

The “Deceptive” test object [Har02] (Figure 5.6) is so named because of the type of objective landscape it imposes with the standard evolutionary approach

```
CFG
Node
 s    void deceptive(double d)
      {
 1        double r;

 2        if (d == 0.0)
 3            r = 0.0;
          else
 4            r = 1.0 / d;

 5        if (r == 0.0)
 6            // target statement
 e    }
```

Figure 5.6: Program code for the "Deceptive" test object

(and the initial event sequence under the hybrid approach) for the coverage of node 6. The test object finds the multiplicative inverse of an input `d`. To avoid a division by zero error, the result of this operation, `r` is zero if `d` is zero. Node 6 is only executed when `r` is zero. However, the zero input value of `d` required to execute the target is unlikely to be found by chance, simply because it represents a very small portion of the overall input domain. The input value of `r` lies in a range of -50,000 to 50,000, with a precision of 0.001, giving a search space size of approximately 10^8. The objective function of the standard evolutionary approach leads the search away from a value of zero for all other input values - since as the value of `d` increases, the value of `r` decreases. This deception can be seen in a plot of the objective function landscape (Figure 5.17a).

Enumeration

This decides whether three inputted colour intensity values (integers in the range 0 to 255) represents one of the colours in an enumeration (Figure 5.7). The target statement (node 26) is executed when the inputs represent the colour black. However plateaux occur in the objective function landscape for the standard evolutionary approach, and the initial event sequence of the hybrid approach (Figure 5.18a), due to the use of a variable that is of the `colour` enumeration type. Because one value from an enumeration cannot be deemed "closer" to any other value from the enumeration (as the ordering of enumeration literals is not significant) plateaux are induced on the landscape in a similar effect to that caused by flag variables. The search space size is approximately

1.6×10^7.

Flag Assignment

This function takes two double values (Figure 5.8). In searching for test data for the execution of node 6, a flag has to be true. The flag is initially set to false, and is only set true when the first input value is zero. The search space is flat for the standard evolutionary approach, and the initial event sequence of the hybrid approach, apart from the point at which the required test data lies (Figure 5.19a). With an input range of -50,000 to 50,000 and a precision of 0.001 for each input variable, the search space size is approximately 10^{16}.

Flag Avoid Assignment

This program is functionally equivalent to "Flag Assignment" (Figure 5.9), except the flag is initially true, and a statement resetting the flag when the first input value is not zero must be avoided. The same input ranges and precision were used.

Flag Avoid Loop Assignment

This is the program of Figure 5.10. The target statement is node 7, which is only executed when all the array values are zero. This is indicated by a flag, which is initially set to true, but is set to false within a loop if any of the array values are found to not be zero. Consequently the search landscape (for the standard evolutionary approach and the initial event sequence of the hybrid approach) is made up of one large plateau (Figure 5.20a) except for the point of the required test data. The range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}.

Flag Loop Assignment

This program (Figure 5.11) also takes an array of ten integer values. A flag is initially set to false, but becomes true when one or more of the array values is zero. This assignment occurs within a loop body. When the flag is true, the target statement (node 8) is executed. Due to the use of the flag, the objective function landscape consists of flat regions (Figure 5.21a), for both the standard evolutionary approach and the initial event sequence under the hybrid approach. The range of the integers of the array was -15,000 to 15,000 giving a search space size of approximately 6×10^{44}.

```
  CFG
 Node
       typedef enum
               {UNKNOWN, WHITE, RED, YELLOW,
                MAGENTA, BLUE, CYAN, GREEN, BLACK} colour;

   s   void enumeration(int r, int g, int b)
       {
   1       colour c;

   2       if (r == 255)
           {
   3           if (g == 255)
               {
  4,5              if (b == 0) c = RED;
  6,7              else if (b == 255) c = MAGENTA;
               }
   8           else if (g == 255)
               {
 9,10              if (b == 0) c = YELLOW;
11,12              else if (b == 255) c = WHITE;
               }
           }
  13       else if (r == 0)
           {
  14           if (g == 0)
               {
15,16              if (b == 255) c = BLUE;
17,18              else if (b == 0) c = BLACK;
               }
  19           else if (g == 255)
               {
20,21              if (b == 0) c = GREEN;
22,23              else if (b == 255) c = CYAN;
               }
           }
  24       else c = UNKNOWN;

  25       if (c == BLACK)
           {
  26           // target statement
           }
   e   }
```

Figure 5.7: Program code for the “Enumeration” test object

```
CFG
Node
 s    void flag_assignment(double a, double b)
      {
 1        int flag = 0;

 2        if (a == 0)
 3            flag = 1;

 4        if (b == 0)
          {
 5            if (flag == 1)
 6                // target statement
          }
 e    }
```

Figure 5.8: Program code for the “Flag Assignment” test object

```
CFG
Node
 s    void flag_avoid_assignment(double a, double b)
      {
 1        int flag = 1;

 2        if (a != 0)
 3            flag = 0;

 4        if (b == 0)
          {
 5            if (flag)
 6                // target statement
          }
 e    }
```

Figure 5.9: Program code for the “Flag Avoid Assignment” test object

```
CFG
Node
 s    void flag_avoid_loop_assignment(int a[10])
      {
 1        int flag = 1;
 2        int i;

 3        for (i = 0; i < 10; i++)
          {
 4            if (a[i] != 0)
 5                flag = 0;
          }

 6        if (flag)
 7            // target statement
 e    }
```

Figure 5.10: Program code for the “Flag Avoid Loop Assignment” test object

```
CFG
Node
 s    void flag_loop_assignment(int a[10], int b)
      {
 1        int flag = 0;
 2        int i;

 3        for (i = 0; i < 10; i++)
          {
 4            if (a[i] == 0)
 5                flag = 1;
          }

 6        if (b == 0)
          {
 7            if (flag)
 8                // target statement
          }
 e    }
```

Figure 5.11: Program code for the “Flag Loop Assignment” test object

```
CFG
Node
 s    void flag_multiple(int r1, int r2)
      {
 1        int error1 = 0;
 2        int error2 = 0;
 3        int shutdown = 0;

 4        if (r1 == 0)
 5            error1 = 1;
 6        if (r2 == 0)
 7            error2 = 1;

 8        shutdown = error1 && error2;

 9        if (shutdown)
10            // target statement
 e    }
```

Figure 5.12: Program code for the “Multiple Flag” test object

Multiple Flag

This program was introduced in Section 5.2.2, and is seen in Figure 5.12. The existence of flags in the program leads to a large plateau in the objective function landscape (Figure 5.22a) for the standard evolutionary approach and the initial event sequence of the hybrid approach.

Both double input ranges were -50,000 to 50,000 with a precision of 0.001 giving a search space size of approximately 10^{16}.

Multiple Flag 2

This program (Figure 5.13) involves two flags, both of which must be true in order to execute the target statement. The program takes two double input variables. Both flags are true if the first and second inputs are zero. However if the second double value is 1, the second flag is reset to false. Both double input ranges were -50,000 to 50,000 with a precision of 0.001, giving a search space size of approximately 10^{16}.

Figure 5.23a shows the plateaux present in the objective function landscape (for both the standard evolutionary approach and the initial event sequence of the hybrid approach), with Figure 5.23b showing the ridge across the plane $a = 0$ to a finer level.

```
CFG
Node
 s    void flag_multiple2(double a, double b)
      {
 1        int initialised = 0;
 2        int has_been_fired = 0;

 3        if (a == 0)
 4            initialised = 1;

 5        if (b == 0)
 6            has_been_fired = 1;

 7        if (initialised)
          {
 8            if (b == 1)
 9                has_been_fired = 1;

10            if (has_been_fired)
11                // target statement
          }
 e    }
```

Figure 5.13: Program code for the “Multiple Flag 2” test object

5.3.2 Experimental Setup

The experiments were conducted in five different setups as detailed below. For each evolutionary search, in both the standard and hybrid approaches, the usual set of search parameters were used (see Section A.4).

Standard Evolutionary Approach

The standard evolutionary approach setup, **StdEA**, uses the node-oriented objective function of Wegener *et al.* [WBS01] (originally introduced in Section 2.3.5). Standard evolutionary approach experiments were conducted in order to demonstrate that the method does indeed encounter problems with the test objects presented.

Usually the evolutionary search is terminated after 200 generations, if no test data has been found. With approximately 300 individuals a generation, this equates to approximately 60,000 trials. In these experiments the evolutionary search continues for up to 2000 generations, equating to approximately 600,000 trials, in order to show that the standard evolutionary approach will not generally find test data just by running the searches for longer.

Hybrid Approach

The hybrid approach was tested in four different setups.

The event sequence generation method was applied in its original form as devised by Ferguson and Korel (referred to as algorithm “1”), and the extended method using influencing sets (referred to as algorithm “2”).

Two different termination criteria are used for the evolutionary searches. Termination criterion “a” terminates searches after 200 generations if no solution has been found. Termination criterion “b”, on the other hand, terminates the search if there has been no improvement in the best objective function value for the last 50 generations. The limit of 50 generations was determined experimentally, as explained in the next section. Criterion “b” can therefore extend the search past the 200 generations limit, providing there has been an improvement in the best objective function value.

The two event sequence generation algorithms and the two different termination criteria for the evolutionary searches were tried in combination with one another to give the following four setups:

- **Hybrid 1a** uses event sequence generation algorithm “1” in combination with evolutionary search termination criterion “a”.

- **Hybrid 1b** uses event sequence generation algorithm “1” in combination with evolutionary search termination criterion “b”.
- **Hybrid 2a** uses event sequence generation algorithm “2” in combination with evolutionary search termination criterion “a”.
- **Hybrid 2b** uses event sequence generation algorithm “2” in combination with evolutionary search termination criterion “b”.

Experimental Measurements

Each experiment was repeated ten times for each different setup and test object. The following information was measured:

- Success rate
- Average number of test data evaluations for a successful search
- Average number of test data evaluations for an unsuccessful search
- Maximum number of test data evaluations for a successful search
- Maximum number of test data evaluations for an unsuccessful search
- Average time for a successful search
- Average time for an unsuccessful search
- Maximum time for a successful search
- Maximum time for an unsuccessful search

Success rate is the percentage of experimental runs that were successful in finding test data for the target statement. For the hybrid setups, the following was also recorded:

- Average number of event sequences considered for a successful search
- Average number of event sequences considered for an unsuccessful search
- Maximum number of event sequences considered for a successful search
- Maximum number of event sequences considered for an unsuccessful search

Table 5.1: Success rate for each experimental setup

	StdEA (%)	**Hybrid 1a** (%)	**Hybrid 1b** (%)	**Hybrid 2a** (%)	**Hybrid 2b** (%)
Counter	0	0	0	100	100
Deceptive	0	100	100	100	100
Enumeration	0	100	100	100	100
Flag Assignment	0	100	100	100	100
Flag Avoid Assignment	0	100	100	100	100
Flag Avoid Loop Assignment	0	80	90	90	100
Flag Loop Assignment	20	100	100	100	100
Flag Multiple	0	0	0	100	100
Flag Multiple 2	0	100	100	100	100

Each experiment was repeated ten times for each different setup and test object. The success rate measure records the percentage of successful attempts that test data was found for each statement by each setup and test object. Search times were measured with a precision of 0.1 of a second.

Experiments were performed on a Pentium 4 PC running Windows XP, with 3GHz and 1Gb RAM under normal load conditions. For further information on the technical details of the experimental framework, see Appendix A.

5.3.3 Results

Table 5.1 shows the percentage of successful runs for each test object and experimental setup. The standard evolutionary approach - as expected - performed very poorly, failing to find test data in all cases apart from two successful attempts with the “Flag Loop Assignment” test object. The hybrid approach performed well, apart from a few exceptions, achieving a 100% success rate for most setups and test objects. For the “Counter” and “Flag Multiple” test objects, the extended event sequence generation algorithm was required. Setups using algorithm “1” could not find test data, whereas a 100% success rate was recorded for setups using algorithm “2”. The hybrid approach did not achieve an overall 100% success rate for the “Flag Avoid Loop Assignment” program. This does not seem to be down to a lack of guidance, but simply due to the evolutionary algorithm not being able to find the test data required in such a large input domain before the search termination criterion was met.

The hybrid approach always finds test data using fewer evaluations than successful and unsuccessful searches using the standard evolutionary setup. This is because the traditional setup usually fails to find test data, and does not terminate until 2000 generations have been evaluated. Normally, the standard evolutionary approach terminates after 200 generations, requiring approximately 60,000 evaluations. The hybrid setups generally require more than 60,000 evaluations because a new evolutionary search is conducted for each event sequence (Table 5.3). However, it was found that the efficiency of the hybrid approach was improved through the use of the new termination criterion (criterion "b") which ends the search after 50 generations of no improvement. This figure of 50 generations was derived by observing the progress of experiments using the usual termination criterion of 200 generations (criterion "a"), which ends the search regardless of recent improvement. Here it was found the majority of unsuccessful searches (e.g. for test data for infeasible event sequences or event sequences where poor guidance was provided to the test data) stagnated well before the 200 generations limit. It was also found that almost all solutions were found within 50 generations of the last improvement of the objective function value (Table 5.2). Only once was this limit broken - for the "Flag Avoid Loop Assignment" program. For this reason termination criterion "b" was employed to simply terminate the search after 50 generations of no improvement. Criterion "b" does not have a detrimental effect on the success rate, as can be seen in Table 5.1. In fact, it improved on the results for "Flag Avoid Loop Assignment" by two more successful runs (i.e. an increase of 20% in the success rate). This was because the number of generations used by the search for an event sequence can in fact extend beyond the 200 generations limit, so long as there has been an improvement in the last 50 generations. Figures 5.14 and 5.15 shows the average number of trials required by each termination criterion (combined for algorithm "1" and "2") for each test object for successful and unsuccessful searches respectively. The bar charts clearly show that criterion "b" is far more efficient that criterion "a", sometimes only using only a quarter of the number of trials.

The results for each individual test object are now discussed in detail.

Counter

Section 5.2.3 explained the process of event sequence generation for the "Counter" test object. The landscape for the initial event sequence:

$$E_0 =< (s, \emptyset), (7\emptyset) >$$

Table 5.2: Average and maximum generations to a solution using termination criterion "a"

Test Object	**Successful Runs**	**Average Generations to Solution**	**Maximum Generations to Solution**
Counter	10	4.4	8
Deceptive	20	11.8	28
Enumeration	20	7.6	34
Flag Assignment	20	6.6	12
Flag Avoid Assignment	20	6.3	19
Flag Avoid Loop Assignment	17	11.8	66
Flag Loop Assignment	20	9.4	48
Flag Multiple	10	3.9	7
Flag Multiple 2	20	5.9	10

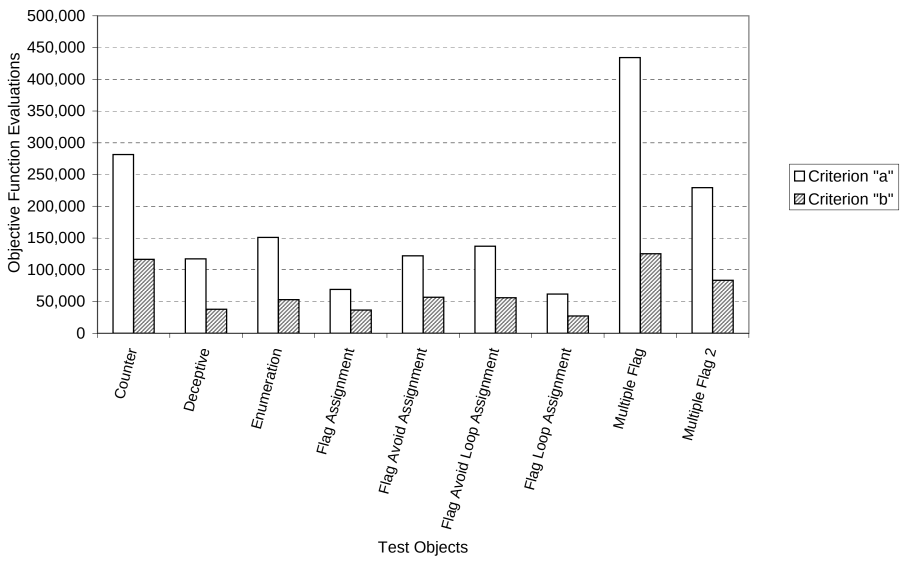


Figure 5.14: Number of objective function evaluations for termination criteria "a" and "b" for successful searches

Table 5.3: Number of objective function evaluations for each test object and experimental setup

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Average	Maximum	Average	Maximum
Counter	StdEA	-	-	547,840	549,128
	Hybrid 1a	-	-	160,445	160,462
	Hybrid 2a	281,551	284,149	-	-
	Hybrid 1b	-	-	44,856	48,743
	Hybrid 2b	116,433	126,806	-	-
Deceptive	StdEA	-	-	534,972	535,547
	Hybrid 1a	117,019	124,198	-	-
	Hybrid 2a	117,276	121,025	-	-
	Hybrid 1b	38,477	40,856	-	-
	Hybrid 2b	37,320	39,255	-	-
Enumeration	StdEA	-	-	534,087	534,087
	Hybrid 1a	151,245	172,931	-	-
	Hybrid 2a	150,699	172,551	-	-
	Hybrid 1b	55,407	62,461	-	-
	Hybrid 2b	50,577	61,359	-	-
Flag Assignment	StdEA	-	-	540,814	543,148
	Hybrid 1a	69,310	73,241	-	-
	Hybrid 2a	68,962	71,479	-	-
	Hybrid 1b	36,392	38,243	-	-
	Hybrid 2b	36,455	40,099	-	-
Flag Avoid Assignment	StdEA	-	-	541,791	543,126
	Hybrid 1a	121,924	124,284	-	-
	Hybrid 2a	121,925	125,674	-	-
	Hybrid 1b	57,050	60,228	-	-
	Hybrid 2b	56,448	58,874	-	-
Flag Avoid Loop Assignment	StdEA	-	-	534,087	534,087
	Hybrid 1a	131,900	143,638	160,448	160,458
	Hybrid 2a	141,926	159,395	160,434	160,434
	Hybrid 1b	59,091	69,636	66,689	66,689
	Hybrid 2b	53,314	61,594	-	-
Flag Loop Assignment	StdEA	184,683	324,408	541,161	545,038
	Hybrid 1a	63,701	66,219	-	-
	Hybrid 2a	59,971	69,670	-	-
	Hybrid 1b	26,292	33,785	-	-
	Hybrid 2b	28,293	35,640	-	-
Flag Multiple	StdEA	-	-	534,087	534,087
	Hybrid 1a	-	-	106,972	106,972
	Hybrid 2a	434,110	435,474	-	-
	Hybrid 1b	-	-	26,820	26,820
	Hybrid 2b	125,185	129,744	-	-
Flag Multiple2	StdEA	-	-	534,087	534,087
	Hybrid 1a	228,967	230,082	-	-
	Hybrid 2a	229,831	231,716	-	-
	Hybrid 1b	82,771	89,399	-	-
	Hybrid 2b	83,902	87,855	-	-

Table 5.4: Search times for each test object and experimental setup

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Average	Maximum	Average	Maximum
Counter	StdEA	-	-	292.4	296.1
	Hybrid 1a	-	-	44.0	44.6
	Hybrid 2a	78.9	79.9	-	-
	Hybrid 1b	-	-	12.9	13.8
	Hybrid 2b	34.5	37.1	-	-
Deceptive	StdEA	-	-	122.51	123.497
	Hybrid 1a	17.3	18.4	-	-
	Hybrid 2a	17.2	17.9	-	-
	Hybrid 1b	6.3	7.3	-	-
	Hybrid 2b	6.5	6.8	-	-
Enumeration	StdEA	-	-	133.1	134.0
	Hybrid 1a	24.5	28.2	-	-
	Hybrid 2a	24.2	27.5	-	-
	Hybrid 1b	9.0	10.9	-	-
	Hybrid 2b	8.9	10.6	-	-
Flag Assignment	StdEA	-	-	136.8	137.4
	Hybrid 1a	12.0	12.5	-	-
	Hybrid 2a	11.7	12.2	-	-
	Hybrid 1b	6.9	7.4	-	-
	Hybrid 2b	6.8	7.4	-	-
Flag Avoid Assignment	StdEA	-	-	136.7	137.8
	Hybrid 1a	20.5	21.2	-	-
	Hybrid 2a	20.3	21.0	-	-
	Hybrid 1b	10.6	11.0	-	-
	Hybrid 2b	10.3	10.8	-	-
Flag Avoid Loop Assignment	StdEA	-	-	295.9	300.3
	Hybrid 1a	37.2	40.5	45.6	45.9
	Hybrid 2a	39.9	44.5	44.7	44.7
	Hybrid 1b	17.8	20.8	19.7	19.7
	Hybrid 2b	16.2	18.3	-	-
Flag Loop Assignment	StdEA	95.8	169.9	296.7	297.8
	Hybrid 1a	17.7	18.3	-	-
	Hybrid 2a	16.6	20.0	-	-
	Hybrid 1b	7.9	9.9	-	-
	Hybrid 2b	8.3	10.3		
Flag Multiple	StdEA	-	-	138.1	138.7
	Hybrid 1a	-	-	15.7	15.8
	Hybrid 2a	39.5	63.6	-	-
	Hybrid 1b	-	-	4.3	4.3
	Hybrid 2b	19.9	20.5	-	-
Flag Multiple2	StdEA	-	-	135.1	135.4
	Hybrid 1a	37.3	37.9	-	-
	Hybrid 2a	36.9	37.5	-	-
	Hybrid 1b	14.9	16.1	-	-
	Hybrid 2b	14.8	15.4	-	-

Table 5.5: Event sequences considered by each setup

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Average	Maximum	Average	Maximum
Counter	Hybrid 1a	-	-	4	4
	Hybrid 1b	-	-	4	4
	Hybrid 2a	7	7	-	-
	Hybrid 2b	7	7	-	-
Deceptive	Hybrid 1a	4	4	-	-
	Hybrid 1b	4	4	-	-
	Hybrid 2a	4	4	-	-
	Hybrid 2b	4	4	-	-
Enumeration	Hybrid 1a	4.7	5	5	-
	Hybrid 1b	5	5	-	-
	Hybrid 2a	4.7	5	5	-
	Hybrid 2b	4.7	5	-	-
Flag Assignment	Hybrid 1a	3	3	-	-
	Hybrid 1b	3	3	-	-
	Hybrid 2a	3	3	-	-
	Hybrid 2b	3	3	-	-
Flag Avoid Assignment	Hybrid 1a	4	4	-	-
	Hybrid 1b	4	4	-	-
	Hybrid 2a	4	4	-	-
	Hybrid 2b	4	4	-	-
Flag Avoid Loop Assignment	Hybrid 1a	4	4	4	4
	Hybrid 1b	4	4	4	4
	Hybrid 2a	4	4	4	4
	Hybrid 2b	4	4	-	-
Flag Loop Assignment	Hybrid 1a	3	3	-	-
	Hybrid 1b	2.8	3	-	-
	Hybrid 2a	2.9	3	-	-
	Hybrid 2b	2.9	3	-	-
Flag Multiple	Hybrid 1a	-	-	3	3
	Hybrid 1b	-	-	3	3
	Hybrid 2a	10	10	-	-
	Hybrid 2b	10	10	-	-
Flag Multiple2	Hybrid 1a	6	6	-	-
	Hybrid 1b	6	6	-	-
	Hybrid 2a	6	6	-	-
	Hybrid 2b	6	6	-	-

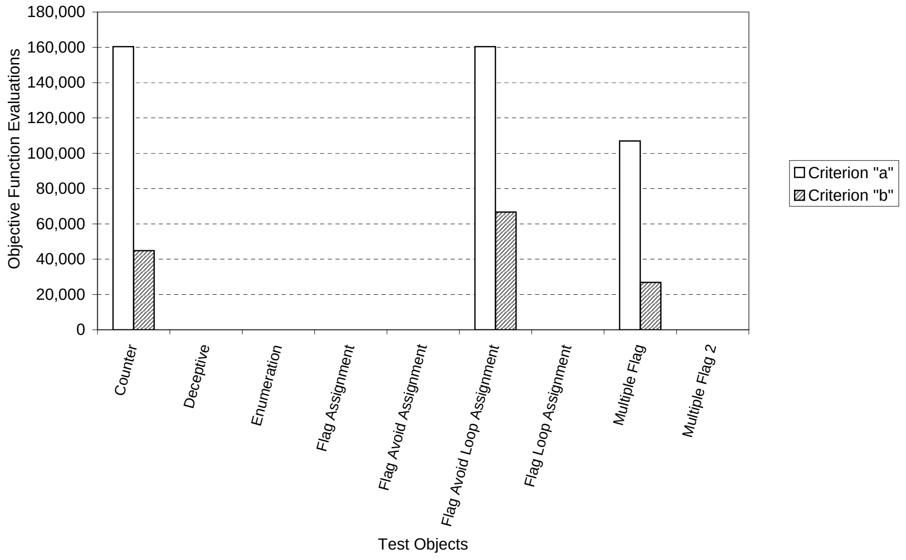


Figure 5.15: Number of objective function evaluations for termination criteria "a" and "b" for unsuccessful searches

and for the standard evolutionary approach contains several plateaux (Figure 5.16a) and the search fails to find test data. Figure 5.16b shows that the search makes some initial progress, but then stagnates.

The chaining algorithm generates event sequences when node 6 is declared as a problem node:

$$\begin{aligned} E_1 &= < (s, \emptyset), (1, \{counter\}), (6, \emptyset), (7\emptyset) > \\ E_2 &= < (s, \emptyset), (5, \{counter\}), (6, \emptyset), (7\emptyset) > \end{aligned}$$

E_2 expects an increment of the `counter` variable, but ridges still appear in the objective function landscape (Figure 5.16c). Again, the search makes some initial progress, but then stagnates and fails (Figure 5.16d). Under the extended algorithm for finding last definitions, further event sequences can be generated. The event sequence:

$$< (s, \emptyset), (5, \{counter\}), (5, \{counter\}), (5, \{counter\}), \\ (5, \{counter\}), (5, \{counter\}), (6, \emptyset), (7\emptyset) >$$

is generated, which forces the path to be taken to target node 7 to increment the value of `counter` five times. This is the minimum number of iterations required for node 7 to become feasible. The objective function landscape for this event sequence is free of plateaux, guiding the search to the required array values so the increment of the `counter` can become possible (Figure 5.16e). With increased levels of guidance, hybrid 2a and 2b setups are able to find the required test data (Figure 5.16f), with seven event sequences considered in total.

Deceptive

All hybrid setups found test data for the target statement of the "Deceptive" test object. The standard evolutionary approach failed to find test data in all attempts. The initial event sequence is:

$$E_0 =< (s, \emptyset), (6, \emptyset) >$$

Initial progress is made as input values of `d` are found that are closer to the negative or positive end of the input domain, and the value of `r` becomes smaller and smaller. However when extreme negative or positive values are found, `r` cannot become any smaller, and the search stagnates. Node 5 is declared as the problem node. Further event sequences are generated:

$$\begin{aligned} E_1 &= < (s, \emptyset), (3, \{r\}), (5, \emptyset), (6, \emptyset) > \\ E_2 &= < (s, \emptyset), (4, \{r\}), (5, \emptyset), (6, \emptyset) > \end{aligned}$$

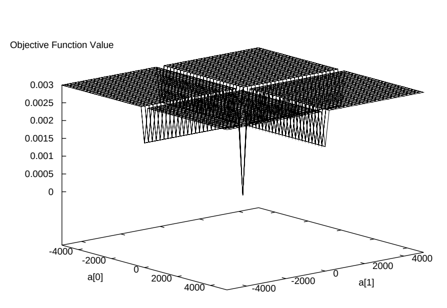
E_1 is feasible and rather than guiding the search away from the desired input value, the objective function landscape guides the search to the required zero value of `d` (Figure 5.17c), and the search succeeds in finding test data (Figure 5.17d).

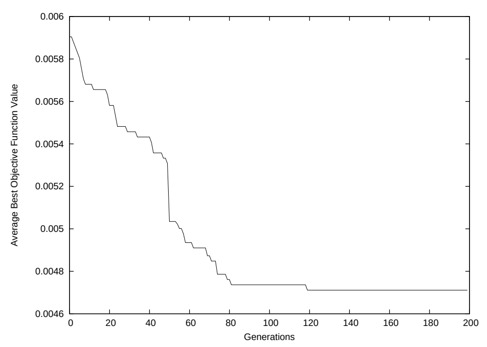
Enumeration

For the "Enumeration" test object, the standard evolutionary approach and the search for test data for the initial event sequence fails because of the lack of guidance provided by the objective function landscape, which is flat due to the use of the enumeration variable in the branch predicate of node 25 (see Figures 5.18a and b). The initial event sequence is:

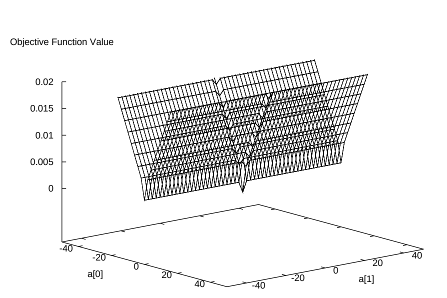
$$E_0 =< (s, \emptyset), (26, \emptyset) >$$

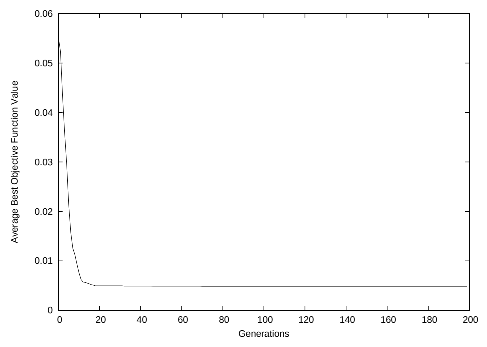
With node 25 declared as a problem node, further event sequences are generated, including:



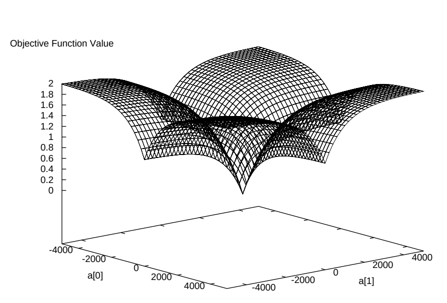


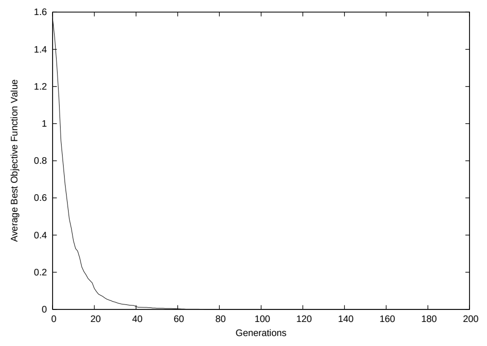
(a) (b)





(c) (d)





(e) (f)

Figure 5.16: Objective function landscapes and average best objective value plots for the “Counter” test object, plotted for the first two values of the array. The next three values are fixed at zero, with the final elements fixed at a non-zero value. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of an intermediate event sequence that requires only one increment of the counter. (d) Average best objective value plot for the search for test data using this intermediate event sequence. (e) Objective function landscape of the final event sequence. (f) Average best objective value plot for the search for test data using the final event sequence.

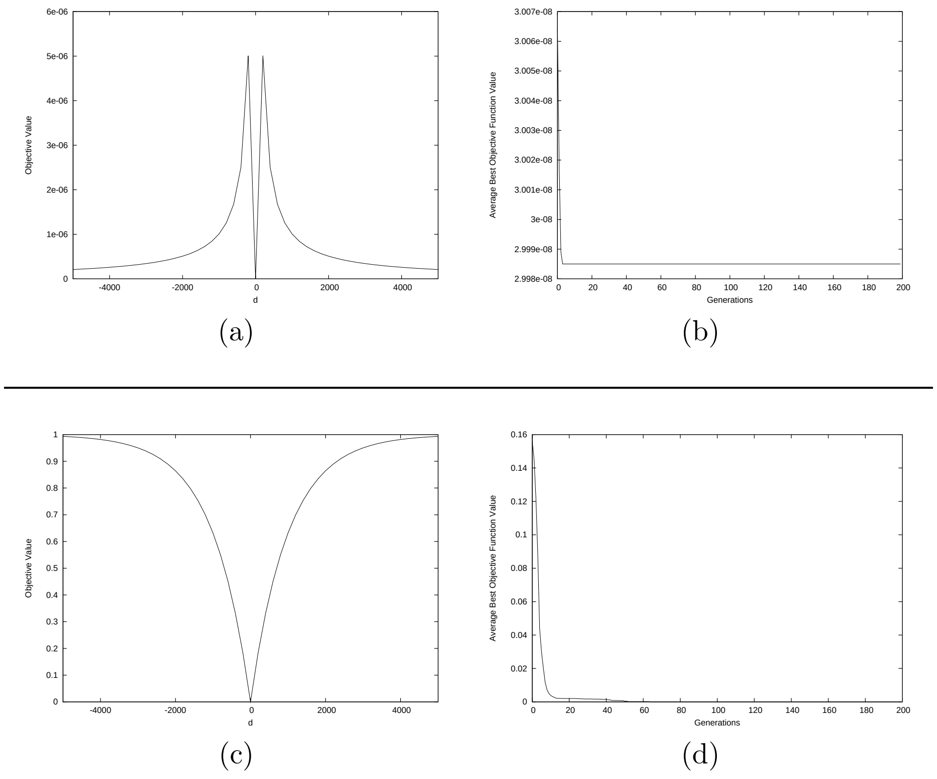


Figure 5.17: Objective function landscapes and average best objective value plots for the "Deceptive" test object. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of the final event sequence. (d) Average best objective value plot for the search for test data using the final event sequence.

$$\begin{aligned}
E_1 &= <(s,\emptyset),(5,\{c\}),(25,\emptyset),(26,\emptyset)> \\
&\ldots \\
E_6 &= <(s,\emptyset),(18,\{c\}),(25,\emptyset),(26,\emptyset)> \\
&\ldots \\
E_8 &= <(s,\emptyset),(23,\{c\}),(25,\emptyset),(26,\emptyset)>
\end{aligned}$$

E_6 is a feasible event sequence providing guidance to the desired values of `r`, `g` and `b`. The plane in the plot of the objective function for this event sequence (Figure 5.18c) appears to be a plateaux. However Figure 5.18d shows it is tilted subtly down to the required input vector. The required test data is successfully found for all runs encountering this event sequence (Figure 5.18e).

Flag Assignment

All hybrid setups found test data for the target statement of the "Flag Assignment" test object. The initial event sequence is:

$$E_0 =< (s,\emptyset),(6,\emptyset) >$$

The flag used at node 5 induces a flat objective function landscape (Figure 5.19a). The search fails to find test data, stagnating very early on in the search (Figure 5.19b). Two event sequences are generated, corresponding to the possible settings of the flag:

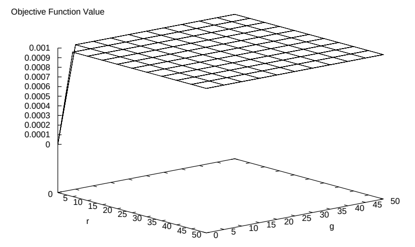
$$\begin{aligned}
E_1 &= <(s,\emptyset),(1,\{flag\}),(5,\emptyset),(6,\emptyset)> \\
E_2 &= <(s,\emptyset),(3,\{flag\}),(5,\emptyset),(6,\emptyset)>
\end{aligned}$$

Node 3 sets the flag to true. The objective function landscape for E_2 is far more amenable to the finding of test data (Figure 5.19d), guiding the search to the input value of `a = 0` which leads to the execution of node 3 (Figure 5.19e).

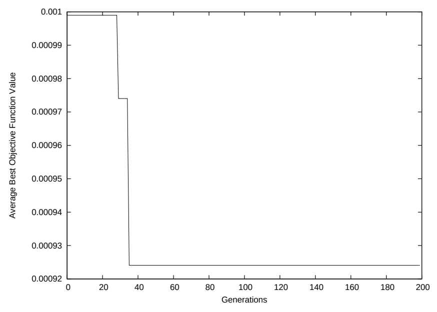
Flag Avoid Assignment

The "Flag Avoid Assignment" test object is similar to "Flag Assignment". The initial event sequence is:

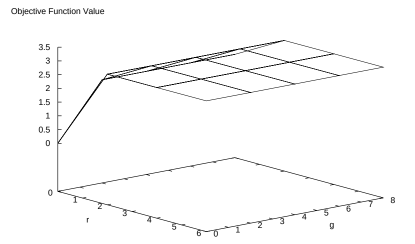
$$E_0 =< (s,\emptyset),(6,\emptyset) >$$



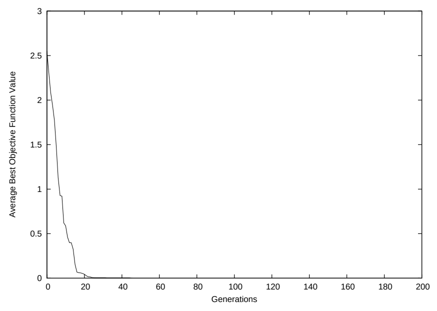
(a)



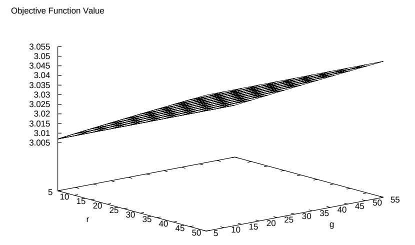
(b)



(c)

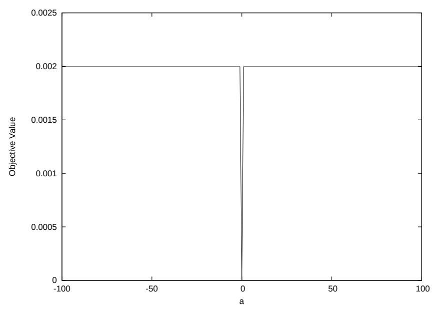


(e)

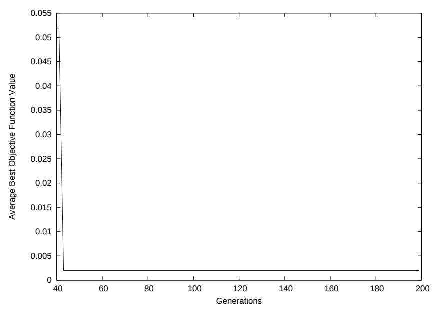


(d)

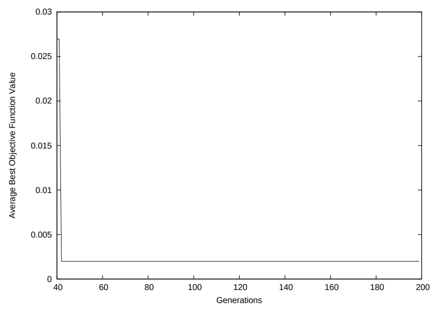
Figure 5.18: Objective function landscapes and average best objective value plots for the "Enumeration" test object. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of the final event sequence. (d) Objective function landscape of the final event sequence cutaway to show slope of plane not apparent in part (c). (e) Average best objective value plot for the search for test data using the final event sequence. Objective function landscapes are plotted for the inputs `r` and `g`, with `b` non-zero.



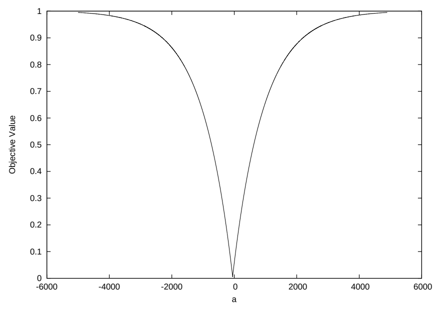
(a)



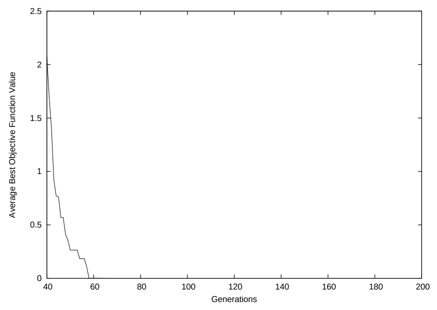
(b)



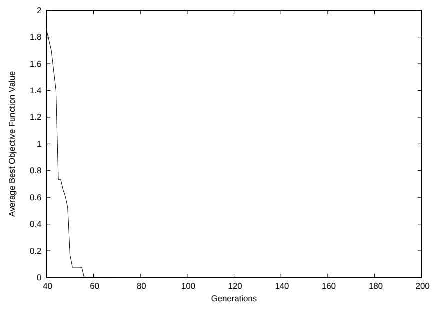
(c)



(d)



(e)



(f)

Figure 5.19: Objective function landscapes and average best objective value plots for the "Flag Assignment" and "Flag Avoid Assignment" test objects. (a) Objective function landscape of the initial event sequence. (b) and (c) Average best objective value plot for the search for test data using the initial event sequence for "Flag Assignment" and "Flag Avoid Assignment" respectively. (d) Objective function landscape of the final event sequence. (e) and (f) Average best objective value plot for the search for test data using the final event sequence for "Flag Assignment" and "Flag Avoid Assignment" respectively.

which has a flat objective function landscape Figure 5.19a, and the search fails (Figure 5.19c). Two event sequences are generated:

$$\begin{aligned} E_1 &= < (s, \emptyset), (1, \{flag\}), (5, \emptyset), (6, \emptyset) > \\ E_2 &= < (s, \emptyset), (3, \{flag\}), (5, \emptyset), (6, \emptyset) > \end{aligned}$$

This time the latter assignment to the flag must be avoided. E_1, therefore, is feasible, whilst E_2 is not. The hybrid approach found the required test data in all cases (Figures 5.19d and f).

Flag Avoid Loop Assignment

The target statement in "Flag Avoid Loop Assignment" has the largest "domain to range" ratio of all the test goals considered in this chapter: from a domain size of approximately 6×10^{44} there is only one input vector which leads to its execution. Even with guidance to avoid the assignment to the flag contained within the loop, the hybrid approach fails to find the required input vector on three of the forty runs. The standard evolutionary approach fails on all attempts.

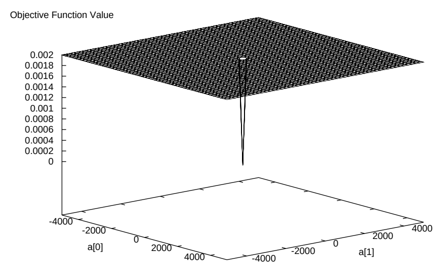
The initial event sequence is:

$$E_0 =< (s, \emptyset), (7, \emptyset) >$$

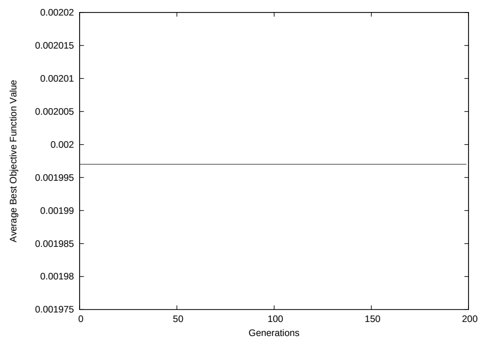
The landscape is flat (Figure 5.20a). Every individual, apart from the input vector where all values of the array of zero, receive the same objective value. Consequently, the search makes no progress (Figure 5.20b). Node 6 is identified as the problem node, with two event sequences generated:

$$\begin{gathered} E_1 =< (s, \emptyset), (1, \{flag\}), (6, \emptyset), (7, \emptyset) > \\ E_2 =< (s, \emptyset), (5, \{flag\}), (6, \emptyset), (7, \emptyset) > \end{gathered}$$

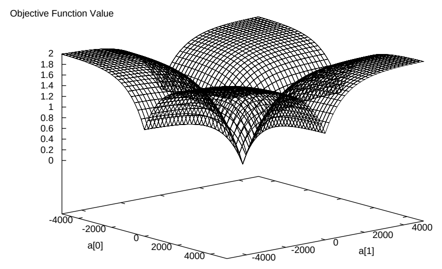
E_1 mandates that the path taken to node 6 avoids the assignment with the loop structure, resulting in a landscape far more conducive to finding test data (Figures 5.20c and d).



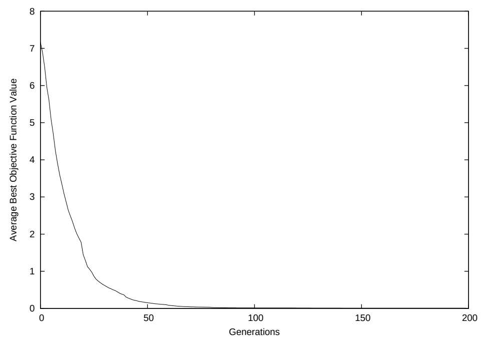
(a)



(b)



(c)



(d)

Figure 5.20: Objective function landscapes and average best objective value plots for the "Flag Avoid Loop Assignment" test object. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of the final event sequence. (d) Average best objective value plot for the search for test data using the final event sequence.

Flag Loop Assignment

The target statement of "Flag Loop Assignment" require one of the inputted array values to be zero, and a secondary input to also be zero for the target statement to be executed. The standard evolutionary approach manages to stumble across such an input vector on two occasions, even though the landscape contains plateaux (Figure 5.21a). The initial event sequence is:

$$E_0 =< (s, \emptyset), (8, \emptyset) >$$

The hybrid approach manages to fortuitously find test data using the initial event sequence on a handful of occasions. When this does not happen, node 7 is found to be a problem node and the following event sequences are generated:

$$E_1 =< (s, \emptyset), (1, \{flag\}), (7, \emptyset), (8, \emptyset) >$$
$$E_2 =< (s, \emptyset), (5, \{flag\}), (7, \emptyset), (8, \emptyset) >$$

E_1 is infeasible, whilst E_2 results in an objective function that provides more guidance to the required test data (Figures 5.21c and d).

Multiple Flag

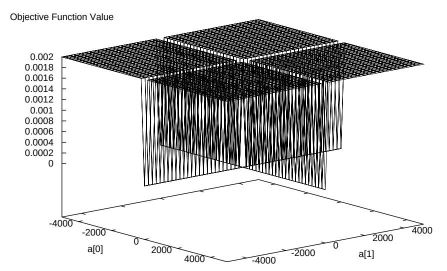
Test data is only found for the "Multiple Flag" test object with the aid of the extended event sequence generation algorithm, as described in Section 5.2.3. Therefore only hybrid setups "2a" and "2b" are successful. Due to the presence of flags, the objective function landscape for the initial event sequence is flat, and the search stagnates (Figures 5.22a and b). The following event sequence is eventually generated by the extended algorithm:

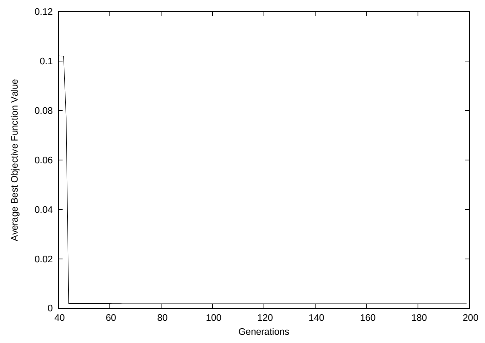
$$< (s, \emptyset), (5, \{error1\}), (7, \{error1, error2\}), (8, \{shutdown\}), (9, \emptyset), (10, \emptyset) >$$

for which the objective function landscape is more conducive to finding test data (Figures 5.22a and b).

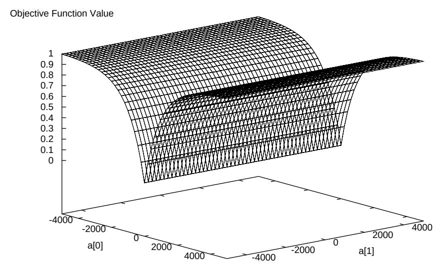
Multiple Flag 2

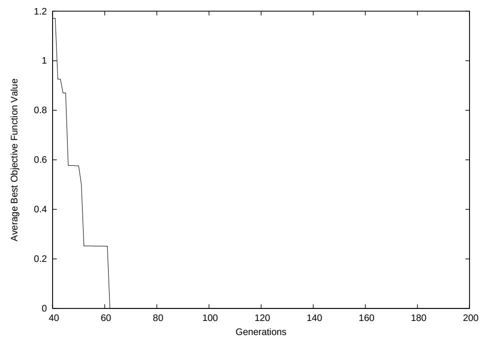
The "Multiple Flag 2" test object also suffers from a flat landscape, due to the use of flags, with two plateaux (Figures 5.23a and b). The test data search fails for the initial event sequence.





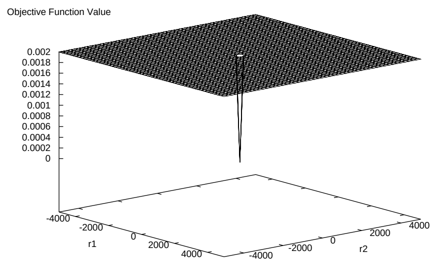
(a) (b)



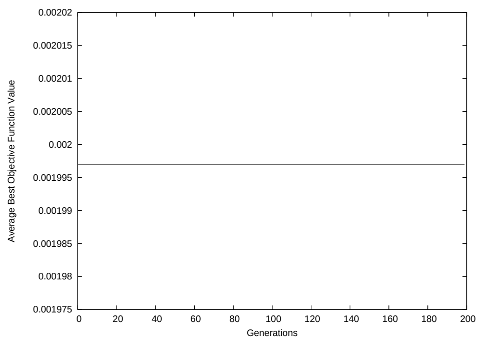


(c) (d)

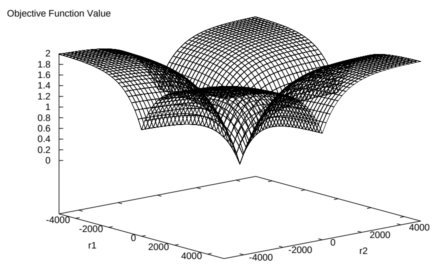
Figure 5.21: Objective function landscapes and average best objective value plots for the "Flag Loop Assignment" test object. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of the final event sequence. (d) Average best objective value plot for the search for test data using the final event sequence.



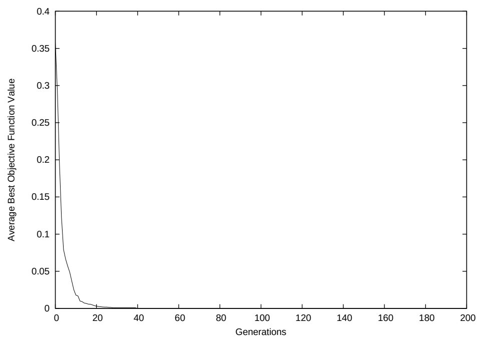
(a)



(b)



(c)



(d)

Figure 5.22: Objective function landscapes and average best objective value plots for the "Multiple Flag" test object. (a) Objective function landscape of the initial event sequence. (b) Average best objective value plot for the search for test data using the initial event sequence. (c) Objective function landscape of the final event sequence. d) Average best objective value plot for the search for test data using the final event sequence.

Either of the following event sequences results in a landscape for which the search can easily find test data:

$$\begin{aligned} E_{2_1} &= <(s,\emptyset),(4,\{initialized\}),(6,\{has\_been\_fired\}),\\ &\qquad (7,\{has\_been\_fired\}),(10,\emptyset),(11,\emptyset)> \\ E_{2_2} &= <(s,\emptyset),(4,\{initialized\}),(7,\emptyset),\\ &\qquad (9,\{has\_been\_fired\}),(10,\emptyset),(11,\emptyset)> \end{aligned}$$

The landscape for E_{2_1} is plotted in Figure 5.23d.

5.3.4 Conclusions of Experimental Study 3

The experimental study shows that the proposed hybrid approach can successfully find test data for programs involving internal variables where the standard evolutionary approach could not. Internal variables other than flags were handled, including counter and enumeration variables.

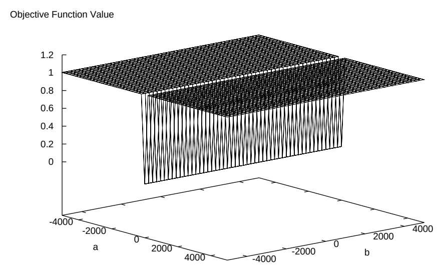
The extensions made to the chaining algorithm enable it to cover statements in the "Counter" and "Multiple Flag" test objects that were not possible with Ferguson and Korel's original algorithm [FK96a].

Furthermore, a new evolutionary search termination criterion was proposed where the search is ended if no improvement is made for 50 generations. This is in contrast to the usual method of terminating the search after 200 generations, whether or not improvements have been made. The new termination criterion was found to be more efficient in the context of several evolutionary searches conducted as part of the hybrid approach. This efficiency gain was achieved without detriment to the success rate of covering test goals.

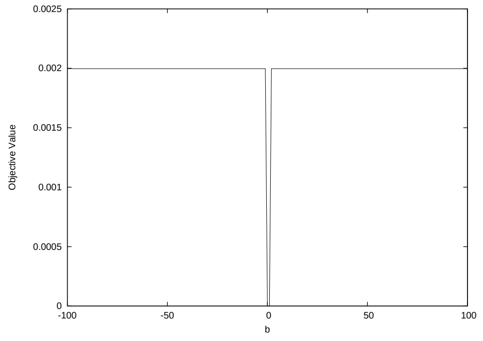
5.4 Conclusions

This chapter has described a novel "hybrid approach" to test data generation, combining evolutionary search with an extended chaining approach. The chaining method used is based on the original approach devised by Ferguson and Korel [FK96a]. The extended method incorporates a number of novel improvements, these being:

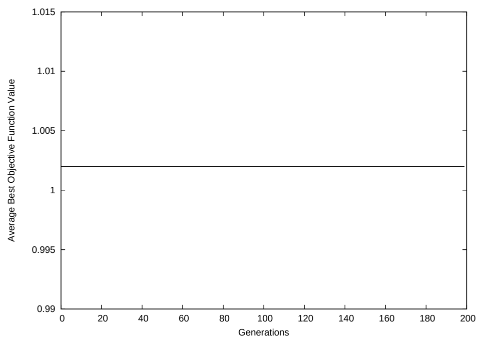
1. The extended chaining method can handle conditions using logical AND and OR connectives, which are extremely common in real-world code. Therefore, the hybrid approach can tackle a wider range of programs than the original approach.



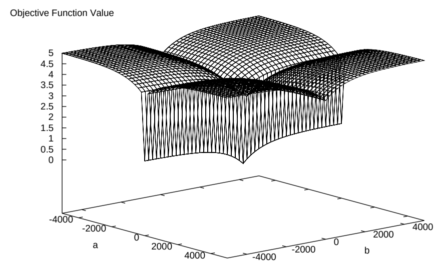
(a)

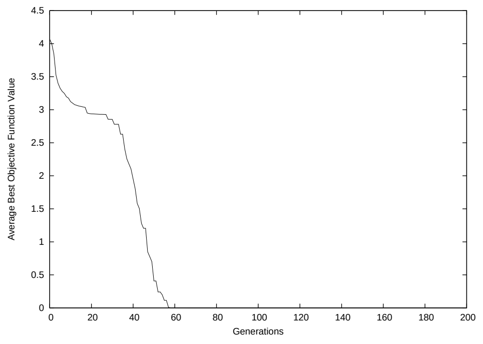


(b)



(c)





(d) (e)

Figure 5.23: Objective function landscapes and average best objective value plots for the "Multiple Flag 2" test object. (a) Objective function landscape of the initial event sequence. (b) Landscape for the initial event sequence again, but for the lower ridge, plotted for $a = 0$. (c) Average best objective value plot for the search for test data using the initial event sequence. (d) Objective function landscape of the final event sequence. (e) Average best objective value plot for the search for test data using the final event sequence.

2. The extended chaining method contains a “return to problem node” capability. If the problem node is encountered more than once by the original chaining approach, it declares failure. The hybrid approach has the option to generate more event sequences which may aid the test data generation process, and overcome the problem node.

3. The extended method features an extended event sequence generation algorithm. This uses the concept of an “influencing set” to identify all variables that can have an influence on the problem node via some program path. Event sequences are generated on the basis of assignments to these variables. The original approach only considers last definition assignments to variables involved in conditions at the problem node.

The hybrid approach was tested with a number of test objects which cause problems for the standard evolutionary structural approach; containing structures which induce landscapes that are flat, coarse or deceptive, due to the use of programming features such as flag, enumeration or counter variables. The hybrid approach with the extended event sequence generation algorithm was able to find test data in all cases. Whilst several authors have attempted to solve the flag problem [Bot02, HHH$^+$02, BS03, BBHK04, HHH$^+$04], there has been almost no work tackling other forms of internal variable problems, such as counters, enumeration variables or internal variables inducing deceptive landscapes.

The next chapter looks at extending the hybrid approach for test objects with state behaviour.

Chapter 6

Extension of the Hybrid Approach for the State Problem

6.1 Introduction

The last chapter introduced a hybrid approach, combining the evolutionary search for test data with an extended chaining approach. This approach works well for input-output functions with structures dependent on internal variables that cause problems for the standard evolutionary approach.

This chapter extends the hybrid approach for test objects with state behaviour. The extended method is used to attempt test data generation for the test objects introduced in Chapter 3, and is referred to as the *sequence hybrid approach.*

6.2 Extending the Hybrid Approach

6.2.1 Finding Last Definitions and Generating Event Sequences

The method for finding last definitions and generating event sequences is unchanged. However, whilst state variables are hidden from calling processes, they are global to the functions concerned. Therefore, last definitions of state variables may not only occur in the same function call as the problem node, but also in prior calls to the function or some other function.

Take the example of Figure 6.1. Suppose the event sequence is:

$$< (s_g, \emptyset), (1, \emptyset), (2, \emptyset) >$$

CFG Node	
s_1	`void global_cfg_example1(int x)`
	`{`
i_1	`    static int initialized = 0;`
1	`    if (initialized)`
2	`        // target node`
3	`    if (x == 0)`
4	`        initialized = 1;`
e_1	`}`

Figure 6.1: Example for finding last definitions for a one function test object with states

where node 1 is a problem node. Since `x` is global to the function, previous definitions for the variable `x` occur at nodes i_1 and 4. Node 4 can only be executed in a previous call to the function.

A similar problem occurs for the execution of the same event sequence for the example of Figure 6.2, except the last definition exists within another function.

So that the event sequence generation procedure can find previous definitions of state variables in prior function calls, a "global" control flow graph is assembled from the individual control flow graphs of each function of the test object. The general form of this control flow graph can be seen in Figure 6.3. The graph uses a special global start node (s_g) and end node (e_g). The global start node has edges leading to a series of state initializing nodes, $i_1 \ldots i_k$, which correspond to the static declarations of the state variables present within the test object. From the final state initializing node i_k are edges leading to the start nodes of the individual functions of the test object $s_1 \ldots s_n$. From each of the end nodes of each function $e_1 \ldots e_n$ are edges leading to the start nodes of each function, along with an additional edge to the global end node.

Event sequences now always begin with the global start node, s_g.

6.2.2 Test Data Search

The test data search for test objects with state behaviour is different to the search used for the hybrid approach for test objects with simple input output behaviour, because a function call sequence needs to be generated. The generation of test data for function call sequences is conducted in two distinct stages.

```
CFG
Node
 i1    static int initialized = 0;

 s1    void global_cfg_example2a()
       {
 1         if (initialized)
 2             // target node

 e1    }

 s2    void global_cfg_example2b(int x)
       {
 3         if (x == 0)
 4             initialized = 1;

 e2    }
```

Figure 6.2: Example for finding last definitions for a multiple function test object with states

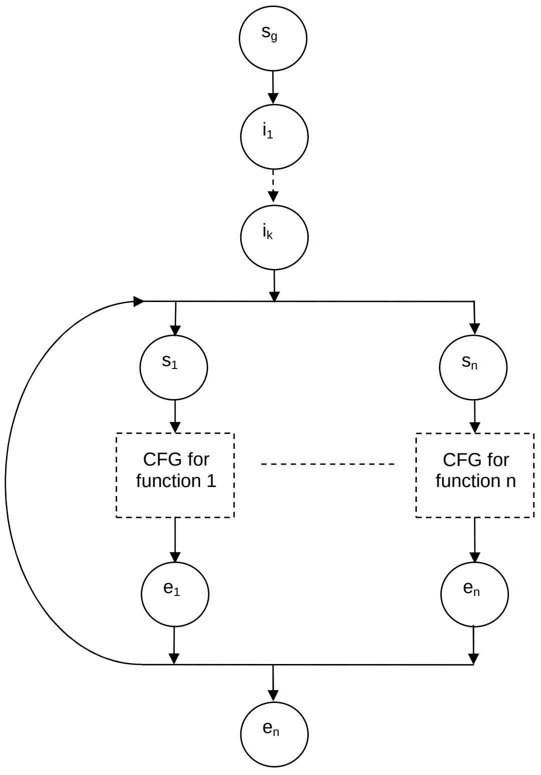


Figure 6.3: “Global” control flow graph for the test objects with state behaviour

Stage One - The Initial Event Sequence

The search for test data for the initial event sequence works in the same manner as for searches using the sequence evolutionary approach described in Section 3.5. The same encoding strategy is used, with a maximum sequence length specified by the tester. Unlike the sequence evolutionary approach, the sequence hybrid approach has the option to extend the sequence length in stage two of the search. Therefore if the value is too small, resulting in function call sequences that are too short, the approach has the possibility of correcting the mistake.

The objective function for stage one works as for the sequence evolutionary approach, attempting to find the minimum objective value for the target node or branch within each call of the sequence.

If no test data can be found that covers the current target structure, the best individual is taken and its initial problem node is identified. If more than one individual has the same best objective value, the individual with the longest function call sequence is preferred.

Stage Two

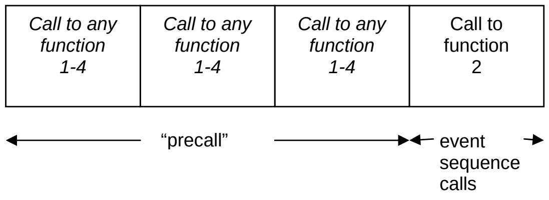
Stage two of the search combines two forms of encoding in two parts. The first part is known as the "precall". Within the precall, the evolutionary search can choose to execute any function in any order for a fixed number of calls. The encoding of this section is identical to that used by the sequence evolutionary approach, using a function call identification number and a universal parameter set (see Section 3.5). The initial length of the precall is the same as the function call sequence length of the best individual from stage one.

The second part of the encoding is referred to as the "event sequence calls" section, and is reserved for the function calls that need to be performed in order to execute each event node in the event sequence. Since the functions to be called are known, no function identification number or universal parameter set is required, and the encoding for each function call is simply based on the parameters to that function.

As an example, take the following event sequence, for the execution of the branch (6, 7) for an arbitrary test object with 4 different functions:

$$< (s_g, \emptyset), (6, \emptyset), (7, \emptyset) >$$

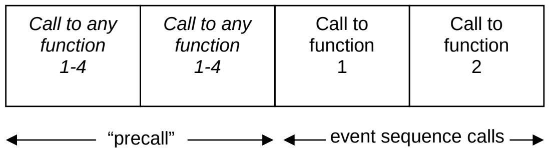
Suppose nodes 6 and 7 lie in function number 2. The initial sequence length is set at 4. The encoding therefore stipulates any function can be called in positions 1-3, whilst function 2 has to be called in position 3:



The length of the precall sequence shrinks as events are added before the original problem node. Suppose node 6 is a problem node, and node 5 is inserted before it in the event sequence:

$$< (s_g, \emptyset), (5, \{var\}), (6, \emptyset), (7, \emptyset) >$$

Suppose node 5 lies in function 1. The precall shrinks to a length of 2:

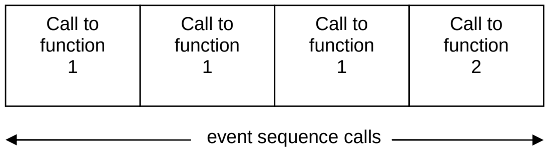


If more functions are required before the initial problem node than the number of calls in the precall, the precall disappears.

Suppose the following event sequence is generated:

$$< (s_g, \emptyset), (5, \{var\}), (5, \{var\}), (5, \{var\}), (6, \emptyset), (7, \emptyset) >$$

The precall section no longer exists, leaving an encoding consisting of function calls for the event sequence only:



The length of the precall for an event sequence is therefore computed using the formula:

$$precall\_len = max(0, init\_seq\_len - init\_prob\_event\_pos)$$

where *precall_len* is the precall length, *init_seq_len* is the sequence length of the best sequence from stage one, and *init_prob_event_pos* is the position of the event in the current sequence that was the first problem node identified from the initial event sequence.

Events can also be inserted between the initial problem node and the event or events corresponding to the target structure. In this case the precall length remains unchanged, but the number of function calls required to execute the event sequence can increase. The functions required to be executed in the event sequence calls portion of the encoding are found by using the following simple algorithm, which takes each pair of adjacent events $e_i = (n_i, C_i)$ and $e_{i+1} = (n_{i+1}, C_{i+1})$ in the event sequence. If n_{i+1} is not reachable from e_n via an acyclic path within the same function, the function parameters are added to the event sequence function calls portion of the encoding.

Take the example of Figure 6.1 once more. If the event sequence is:

$$< (s_g, \emptyset), (4, \{initialized\}), (1, \emptyset), (2, \emptyset) >$$

The function call sequence is:

$$< global\_cfg\_example1(), global\_cfg\_example1() >$$

Node 4 is not reachable from s_g without invoking $global\_cfg\_example1()$, and so the first call is added. Node 1 is not reachable from node 4 without another call to the function, resulting in the second call. Node 2 is reachable from node 1 without the need for another function call, and so no further function calls need to be inserted.

6.2.3 Chaining Tree Search

In contrast to all previous experiments using the chaining approach, the chaining tree is explored in a breadth-first manner for test objects with states. Due to the fact that event sequences may be very long as well as relatively short, the fixed depth limit on the chaining tree for a depth-first search would have to be relatively high. This would mean the chaining tree search for structures requiring only short event sequences may take a long time, due to the fact that wrong branches were chosen to be expanded down to the depth limit. With a breadth-first search this does not happen, as each level of the tree is fully explored before considering the next.

6.3 Experimental Study 4 - Test Data Generation using the Sequence Hybrid Approach

6.3.1 Experimental Setup

Using the sequence hybrid approach, full branch coverage was attempted for the same set of test objects described in Section 3.3.1 with the same initial sequence lengths as those used for the sequence evolutionary approach experiments (Table 3.5). Instead of fixing a maximum depth as the termination criterion for the chaining tree search, a fixed limit of 200 event sequences was imposed. Each evolutionary search for each event sequence was terminated after 50 generations of no improvement. This setup is referred to as the **SeqHybrid** setup.

Random sequence generation experiments were repeated. This time, the number of random test sequence evaluations that can be used for each branch was equivalent to the highest number of evaluations considered by successful or unsuccessful sequence hybrid approach searches for that branch. This setup is referred to as the **SeqRnd** setup.

Each branch is taken as the individual target of the search, regardless of whether it was fortuitously covered during the test data search for another branch. The search for each branch with each setup was repeated ten times. Once again the following information was measured:

- Success rate
- Coverage
- Average number of test data evaluations for a successful search
- Average number of test data evaluations for an unsuccessful search
- Maximum number of test data evaluations for a successful search
- Maximum number of test data evaluations for an unsuccessful search
- Average time for a successful search
- Average time for an unsuccessful search
- Maximum time for a successful search
- Maximum time for an unsuccessful search

For definitions of success rate, coverage, and other measurements, see Section 3.3.2.

Table 6.1: Success rate and coverage using the sequence hybrid approach, compared to random test sequence generation

Test Object	**Test Method**	**Success Rate (%)**	**Coverage (%)**
Anomaly Detector	SeqHybrid	100.0	100.0
	SeqRnd	14.3	14.3
Array Difference	SeqHybrid	100.0	100.0
	SeqRnd	91.7	91.7
Postcode	SeqHybrid	62.3	87.9
	SeqRnd	26.7	31.8
Sliding Window	SeqHybrid	100.0	100.0
	SeqRnd	36.9	46.2
Smoke Detector	SeqHybrid	91.4	100.0
	SeqRnd	71.4	71.4
Sortcode	SeqHybrid	93.7	100.0
	SeqRnd	76.9	88.4
Stack	SeqHybrid	100.0	100.0
	SeqRnd	75.0	75.0
Tel No	SeqHybrid	88.0	94.1
	SeqRnd	42.6	52.9
Vending Machine	SeqHybrid	100.0	100.0
	SeqRnd	93.4	93.8

6.3.2 Results

Table 6.1 shows the success rate and coverage levels for the sequence hybrid approach. The sequence hybrid approach achieved full coverage for 7 of the 9 test objects. A 100% success rate is achieved for five out of the nine test objects. The approach achieves higher coverage and success rates than the random approach over a similar number of test sequence evaluations in all cases.

These results are compared to those obtained with the standard and sequence evolutionary approaches from Chapter 3 in Figures 6.4 and 6.5. The sequence hybrid approach achieves the highest coverage levels and success rates in all cases. The sequence hybrid approach generally performs a higher number of evaluations (Table 6.2) than both standard and sequence evolutionary approaches. However, increasing the number of generations (and consequently the number of evaluations) from 200 to 1000 for both of these approaches only resulted in a marginal improvement in coverage and success rate. It is unlikely that extending the number of generations even further for both approaches

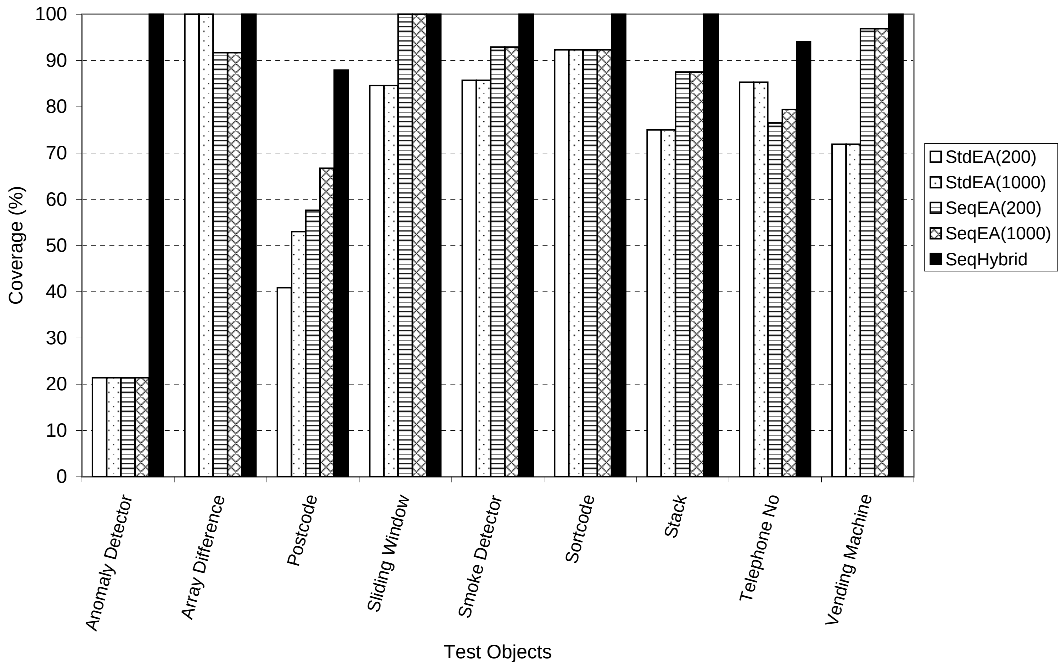


Figure 6.4: Comparison of coverage levels for the sequence hybrid approach against the standard and sequence evolutionary approaches. For details of the standard (StdEA) and sequence (SeqEA) evolutionary approach experiments, refer to Chapter 3

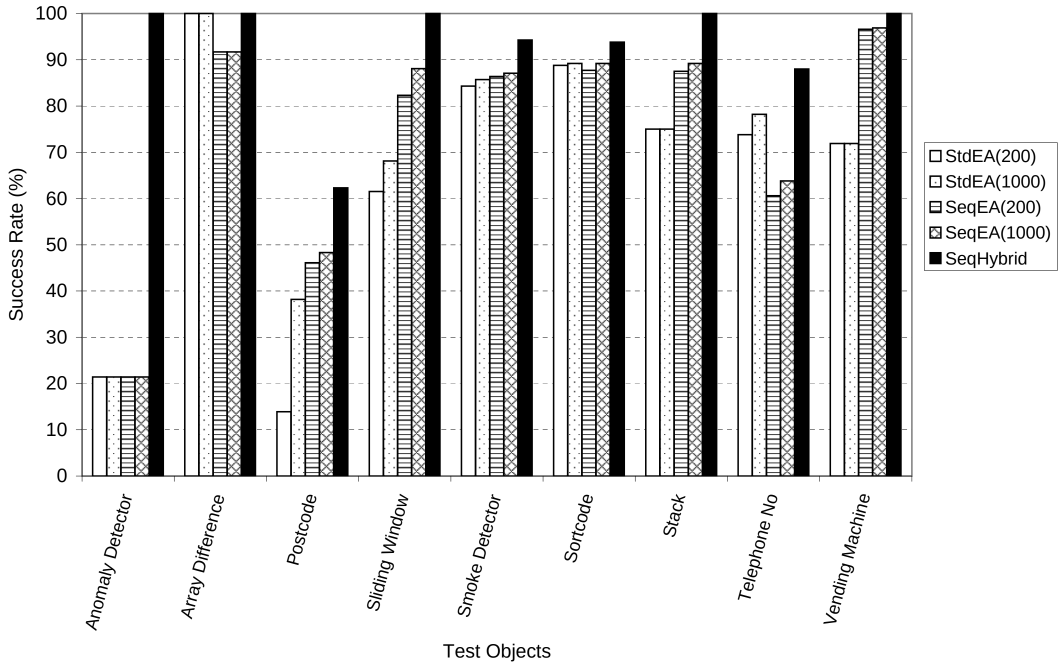


Figure 6.5: Comparison of success rate levels for the sequence hybrid approach against the standard and sequence evolutionary approaches. For details of the standard (StdEA) and sequence (SeqEA) evolutionary approach experiments, refer to Chapter 3

Table 6.2: Average and maximum number of test sequence evaluations for searches using the sequence hybrid approach and random sequence generation

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Av Evals	Max Evals	Av Evals	Max Evals
Anomaly Detector	SeqHybrid	100,733	626,197	-	-
	SeqRnd	1	1	129,285	626,197
Array Difference	SeqHybrid	9,532	117,022	-	-
	SeqRnd	188	4,237	117,022	117,022
Postcode	SeqHybrid	303,504	8,958,755	2,863,592	14,403,980
	SeqRnd	85,157	1,080,520	4,102,282	14,403,980
Sliding Window	SeqHybrid	7,701	63,649	-	-
	SeqRnd	555	26,586	21,058	63,649
Smoke Detector	SeqHybrid	98,526	3,438,822	3,900,739	4,102,926
	SeqRnd	1	1	2,021,827	4,102,926
Sortcode	SeqHybrid	53,452	3,216,321	3,490,579	5,402,062
	SeqRnd	4,710	94,605	2,502,847	5,402,062
Stack	SeqHybrid	4,249	36,739	-	-
	SeqRnd	1	2	23,132	36,739
Tel No	SeqHybrid	215,506	7,269,153	3,210,451	13,288,663
	SeqRnd	1,144	19,376	2,153,902	13,288,663
Vending Machine	SeqHybrid	1,090	31,486	-	-
	SeqRnd	9	176	18,078	31,486

would result in any dramatic improvement, especially on the basis of data recorded in Chapter 3, Table 3.9, which suggests that on average, unsuccessful searches stagnated after the 100th generation, making no further progress. The sequence hybrid approach makes use of extra test data evaluations by searching using different event sequences (refer to Table 6.4), guiding the search into different areas of the test object's input domain. Table 6.5 shows the maximum and average chaining tree depth explored for searches using the hybrid approach. The concept of the chaining tree was defined on page 114.

The question now addressed is why 100% coverage and success rates were not achieved for all test objects. This is down to the following two reasons:

Exponential Growth of the Chaining Tree

Some branches were not consistently covered, or even covered at all, because the chaining tree simply became too large for all feasible event sequences to be explored before the 200 event sequence termination limit.

For the "Smoke Detector" test object, experiments were re-run ten times

Table 6.3: Average and maximum times for searches using the sequence hybrid approach and random sequence generation

Test Object	**Test Method**	**Successful Search**		**Unsuccessful Search**	
		Av Time (s)	Max Time (s)	Av Time (s)	Max Time (s)
Anomaly Detector	SeqHybrid	143.0	946.8	-	-
	SeqRnd	0.1	0.1	24.4	120.1
Array Difference	SeqHybrid	10.5	106.2	-	-
	SeqRnd	0.1	0.7	17.2	17.3
Postcode	SeqHybrid	176.9	6,551.0	1,869.5	11,643.9
	SeqRnd	12.1	153.4	580.6	2,096.0
Sliding Window	SeqHybrid	5.3	24.3	-	-
	SeqRnd	0.1	2.2	1.7	5.2
Smoke Detector	SeqHybrid	56.9	2,116.4	2,512.3	2,668.2
	SeqRnd	0.1	0.1	488.5	995.1
Sortcode	SeqHybrid	44.5	3,339.0	3,624.7	5,983.0
	SeqRnd	1.3	26.2	685.5	1,496.6
Stack	SeqHybrid	5.7	35.3	-	-
	SeqRnd	0.1	0.1	4.5	7.5
Tel No	SeqHybrid	146.1	6,024.8	2,171.5	10,328.8
	SeqRnd	0.3	4.2	462.7	2,899.2
Vending Machine	SeqHybrid	0.9	11.8	-	-
	SeqRnd	0.1	0.1	1.4	2.3

Table 6.4: Event sequences used for sequence hybrid approach searches. “Av ES” refers to the average number of event sequences used per branch, “Max ES” refers to the maximum number of event sequences used for a branch

Test Object	**Successful Search**		**Unsuccessful Search**	
	Av ES	Max ES	Av ES	Max ES
Anomaly Detector	6.4	31	-	-
Array Difference	1.3	4	-	-
Postcode	8.2	141	66.5	200
Sliding Window	1.3	6	-	-
Smoke Detector	6.1	175	200	200
Sortcode	3.2	131	140	200
Stack	1.1	2	-	-
Tel No	6.8	158	80.1	200
Vending Machine	1.1	3	-	-

Table 6.5: Chaining tree depth reached by sequence hybrid approach searches

Test Object	**Successful Search**		**Unsuccessful Search**	
	Av Depth	Max Depth	Av Depth	Max Depth
Anomaly Detector	1.6	6	-	-
Array Difference	0.2	2	-	-
Postcode	1.1	11	8.1	15
Sliding Window	0.3	2	-	-
Smoke Detector	0.3	6	6.4	7
Sortcode	0.4	10	9.6	15
Stack	0.1	1	-	-
Tel No	1	13	4.6	10
Vending Machine	0	1	-	-

with an imposed limit of 1000 event sequences, and test data was generated with a 100% success rate. The 200 event sequence limit was broken 12 times. The number of event sequences attempted reached a maximum total of 622 for one particular branch, using approximately 13 million test data evaluations and a search time of almost three hours.

The 200 event sequence limit was also reached on 5, 10 and 15 occasions for the “Postcode”, “Sortcode” and “Telephone No” test objects respectively.

Nesting and Composed Conditions

Event sequences can consist of several nodes which lie within nested structures, or nested within composed conditions. Therefore, the method is particularly susceptible to the problems of nesting and short-circuiting due to the use of the `&&` and `||` logical operators described in Section 2.3.5, page 52, because if test data can not be found to execute every node correctly in the event sequence, the search will have failed. In this way feasible event sequences might be considered for which test data may not always be found.

This problem hampered test data generation for the “Postcode”, “Sortcode” and “Telephone Number” test objects, where target branches required sequences of nested nodes. The “Postcode” test object in particular has a high level of nesting, and suffered the most problems. Full coverage was achieved with the “Sortcode” test object, but not with a 100% success rate. The “Sortcode” test object has some nodes with a nesting level of 4, as well as a number of conditions composed using the `&&` operator. The “Telephone Number” test object has a high number of composed conditions. In eleven of its `if` statements, seven are constructed using the `&&` and `||` operators.

6.4 Experimental Study 5 - An Industrial Test Object

An experiment was undertaken on a car controller module provided by Daimler-Chrysler. The module contains one principal function, and two auxiliary static functions.

The main function has fifteen inputs. Two variables were of the `double` type, the first in the range -30.0 to 30.0 with a precision of 0.001, and the second in the range -30.0 to 100.0, also with a precision of 0.001. The remaining thirteen variables were boolean inputs, modelled using the `unsigned integer` type in the range 0 to 1. It contains thirteen internal `static` state variables, with a further eight state variables declared outside the function.

The main function has approximately 425 lines of code, with 61 if-then statements resulting in 122 branches, and a maximum nesting depth of 10.

It was discovered within the code that there were three assignments to boolean variables using the result of a boolean expression, i.e.:

```
boolean_var = (boolean_expression)
```

In such situations, the chaining approach cannot guide the search to either a specific true or false outcome of the boolean flag variable, which might be important for the coverage of some branch. Therefore, these assignments were converted into decisions of the form:

```
if (boolean_expression)
    boolean_var = true;
else
    boolean_var = false;
```

This increased the number of decisions in the code by three, making 128 branches in total.

The final code was then tested using the sequence evolutionary approach with a termination criterion of 200 generations, and also with the sequence hybrid approach. Each branch is taken as the individual target of the search, regardless of whether it was fortuitously covered during the search for test data for another branch. The search for each branch was repeated for each method

Table 6.6: Success rate and coverage for the industrial experiment

Test Method	**Success Rate (%)**	**Coverage (%)**
SeqEA(200)	93.1	96.1
SeqHybrid	96.3	96.9

Table 6.7: Evaluations for the industrial experiment

Test Method	**Successful Search**		**Unsuccessful Search**	
	Av Evals	Max Evals	Av Evals	Max Evals
SeqEA(200)	991	40,188	56,175	75,789
SeqHybrid	4,121	214,357	5,639,585	8,232,486

five times.

6.4.1 Results

Table 6.6 shows the success rate and coverage for the experiment. The sequence hybrid approach achieved a higher coverage and success rate than the sequence evolutionary approach.

The sequence evolutionary approach could not cover five branches. Of these the sequence hybrid approach failed to cover four, with the fifth covered in one of the runs. Two of the four branches not covered were infeasible. The sequence hybrid approach could not find test data for the remaining two branches, even with the use of chaining. Again, this was down to chaining tree explosion problems, along with nesting and short-circuit operators preventing the search from finding test data for feasible event sequences.

The sequence evolutionary approach failed to cover ten branches consistently, resulting in a lower success rate. These branches were always covered by the sequence hybrid approach, due to the use of chaining. In all five runs,

Table 6.8: Times for the industrial experiment

Test Method	**Successful Search**		**Unsuccessful Search**	
	Av Time	Max Time	Av Time	Max Time
SeqEA(200)	3.6	111.5	142.6	156.8
SeqHybrid	9.2	423.3	14,005.5	28,390.2

Table 6.9: Event sequences attempted in the industrial experiment

Successful Search		**Unsuccessful Search**	
Average	Maximum	Average	Maximum
1.2	14	200	200

Table 6.10: Chaining tree depth reached for the industrial experiment

Successful Search		**Unsuccessful Search**	
Av Depth	Max Depth	Av Depth	Max Depth
0.1	2	5.5	7

chaining was used on 56 occasions, with the method successfully finding test data in 32 of those attempts. 10 of the failed 24 attempts corresponded to searches for test data for infeasible branches.

6.5 Conclusions

The sequence hybrid approach successfully improves on the sequence evolutionary approach, obtaining higher coverage levels with a greater success rate. In many cases, 100% coverage was obtained, and in the majority of these cases with a 100% success rate. Problems still exist with the exponential growth of the chaining tree and the number of event sequences that have to be considered. Another difficulty exists where the problems of nested structures and short-circuit condition evaluation are accentuated due to the number of nodes for which test data must be found to execute in turn.

An experiment was carried out with a real-world test object. The sequence hybrid approach proved to generate test data more reliably, with a higher success rate. Due to the coverage of one problematic branch in one experimental run, the sequence hybrid approach also achieved a slightly higher coverage rate.

Chapter 7

Conclusions and Future Work

7.1 Summary of Achievements

The original overall aims and objectives of this thesis were as follows:

1. To identify the problems that test objects with state behaviour can cause for evolutionary test data generation for procedural programs written in the C language; and

2. To propose extensions to the evolutionary test data generation framework so that test data generation might be improved.

7.1.1 Identification of Problems for the Standard Evolutionary Approach and State-Based Test Objects

Chapter 3 identified the two main problems that state-based test objects pose for the standard evolutionary approach; these being:

- the need for input sequences; and

- the problem of internal variables.

Input sequences need to be generated for the coverage of certain structures, which require the test object to be put into some state. The internal variable problem causes difficulties for test objects with state behaviour and input-output behaviour. Their use in state-based test objects, however, is inevitable, since internal variables are required to manage state information. The use of internal variables in the conditions of programs can result in a degree of "information loss" when computing the branch distance measure, producing coarse or flat objective function landscapes for structures within the program.

Table 7.1: Description of evolutionary test data generation methods used in this thesis

Test method	Description	Chapter	Novel to thesis
Standard evolutionary approach	Test data generation of inputs for atomic function calls for test objects with input-output behaviour	2/3	No
Sequence evolutionary approach	Test data generation of input sequences of function calls for test objects with state behaviour	3	Multiple function version
Hybrid approach	Hybrid evolutionary-extended chaining approach for test data generation of inputs for atomic function calls for test objects with input-output behaviour	5	Yes
Sequence hybrid approach	Hybrid evolutionary-extended chaining approach for generating inputs sequences of function calls for test objects with state behaviour	6	Yes

This in turn results in the search receiving less guidance to the required test data, and may possibly fail.

7.1.2 Solution Development for Improving Evolutionary Test Data Generation for State-Based Test Objects

The solution proposed to the state problem was developed in three stages. The different methods are described, in brief, in Table 7.1. The first stage tackled the problem of generating input sequences for test objects involving one or more callable functions, for example as part of a module. The resultant method is referred to as the *sequence evolutionary approach*.

The second stage tackled the problem of internal variables by hybridization of the standard input-output evolutionary approach with the *chaining approach* for test objects with input-output behaviour. The resultant method is referred to as the *hybrid approach*.

The third stage brought the first two components together in an attempt to solve the state problem as a whole (Figure 7.1). The resultant method is referred to as the *sequence hybrid approach*.

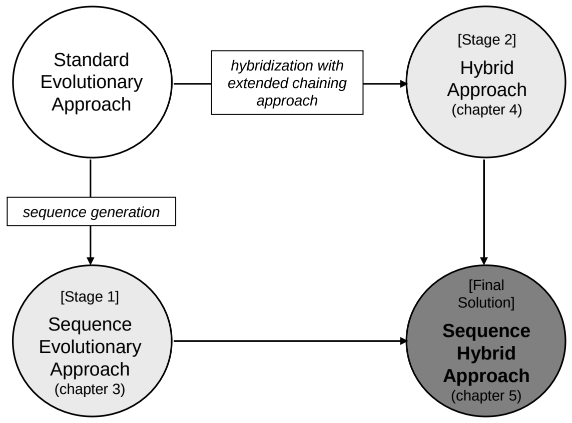


Figure 7.1: Development of the solution proposed by this thesis to the state problem

7.1.3 Stage One - Sequence Evolutionary Approach

Chapter 3 introduced an encoding for the generation of input sequences for test objects consisting of a single callable function devised by Baresel *et al.* [BPS03]. This was extended to handle test objects with multiple functions composed as part of modules. Experiments were performed with nine synthetic test objects with state behaviour. The sequence method achieved the same or higher branch coverage levels than the standard evolutionary approach for seven of the nine test objects. The standard evolutionary approach was only able to cover structures dependent on the state of the test object because the test object is not reset during the evaluation of a generation of atomic function calls. Therefore the "input sequence" found can be as long as the number of individuals in a generation, rather than sequences of length specified by the tester, as for the sequence approach. The standard evolutionary approach cannot cover certain structures in test objects with multiple callable functions.

7.1.4 Stage Two - The Hybrid Approach

The hybrid approach was proposed in Chapter 5 to overcome internal variable problems for test objects with internal variables. The approach is a hybridization of the standard evolutionary approach with the chaining approach [FK96a, Kor96, FK96b]. The basic idea of the chaining approach is to find a sequence of statements involving internal variables that need to be executed prior to the test goal. By requiring that these statements are executed, information previously unavailable to the search can be utilized, possibly guiding it into potentially promising and unexplored areas of the test object's input domain.

The chaining method used is based on the original approach devised by Ferguson and Korel [FK96a]. The extended method incorporates a number of novel improvements, these being:

1. The extended chaining method can handle conditions using logical AND and OR connectives, which are extremely common in real-world code. Therefore, the hybrid approach can tackle a wider range of programs than the original approach.

2. The extended chaining method contains a "return to problem node" capability. If the problem node is encountered more than once by the original chaining approach, it declares failure. The hybrid approach has the option to generate more event sequences which may aid the test data generation process, and overcome the problem node.

3. The extended method features an extended event sequence generation algorithm. This uses the concept of an "influencing set" to identify all variables that can have an influence on the problem node via some program path. Event sequences are generated on the basis of assignments to these variables. The original approach only considers last definition assignments to variables involved in conditions at the problem node.

For nine test objects with simple input-output behaviour, test data could be found for a specific "problem" statement in each test object with the hybrid approach. Test data was not found for eight problem statements with the standard evolutionary approach, and only twice in ten attempts for the remaining test object problem statement. Whilst several authors have attempted to solve the flag problem [Bot02, HHH^{+}02, BS03, BBHK04, HHH^{+}04], there has been almost no work tackling other forms of internal variables problems, such as counters, enumeration variables or internal variables inducing deceptive landscapes.

7.1.5 Final Stage - The Sequence Hybrid Approach

The sequence hybrid approach, introduced in Chapter 6, combines elements of the sequence and hybrid approaches. For the original nine state-based test objects for which experiments were carried out with the standard and sequence evolutionary approaches, higher levels of branch coverage was obtained with the sequence hybrid approach. Full branch coverage was obtained with five of the test objects, and a 100% success rate in generating test data for all branches over all search repetitions was achieved for four test objects.

A further experiment was conducted with a real-world industrial test object. When comparing the sequence evolutionary approach with the sequence hybrid approach, the sequence hybrid approach achieved a slightly higher coverage rate, and a higher rate of success in finding test data for all branches in all five experimental runs.

7.1.6 Overall Conclusions

This thesis has identified the problems that test objects with state behaviour can cause for the standard evolutionary approach to evolutionary test data generation. A solution was developed in three stages to handle these problems.

A number of experiments show the value of the approach for both test objects with states, and also test objects with simple input-output behaviour. In all the test objects considered, higher levels of branch coverage are obtained, improving on previous results with the standard evolutionary approach.

7.2 Restrictions and Limitations

This section outlines some of the restrictions and limitations of the approach developed.

7.2.1 Types of Programs

The method is currently limited to procedural test objects written in the C language.

The scope of the data flow analysis for the generation of event sequences is currently limited, in that program code utilizing dynamic memory through the use of pointers cannot be handled. Thus test data can be generated for individual modules, but not multiple instances of abstract data types.

The method is inhibited by high levels of nesting and the presence of short-circuiting logical operators in programs. This is a general issue for evolutionary test data generation, as discussed in Section 2.3.5. The approach developed requires sequences of statements to be executed, and if many of these statements are deeply nested or depend on the outcome of composed conditions using the `&&` and `||` operators, there are more opportunities for these problems to inhibit the test data search.

Programs with side-effects cannot be handled, as the event sequence model can only handle one definition per event. Side-effects could be removed via a side-effect removal transformation [HHZM01].

Furthermore, the approach has not been tested with unstructured programs, containing forward or backward jumps, for example through the use of `break` or `goto` statements. Handling unstructured programs is not a major theoretical problem for the approach, but not one that has been addressed.

7.2.2 Scalability

The method, as it currently stands, has some issues in the area of scalability. For some programs, the chaining tree grows too large to be explored exhaustively. This prevented full coverage from being obtained for a small number of the test objects in the experiments. This is particularly problematic when the test object's code has a relatively high number of definitions for each variable, meaning more event sequences are generated, or have recursive definitions for which a variable is both defined and used in the same statement. The latter can cause the same node to appear many times in the event sequence. The next section describes how these problems might be circumvented.

7.3 Summary of Future Work

The approach currently has some limitations with respect to the type of programs that can be handled, and issues with scalability. These need to be addressed. Detailed areas for future work along these lines are described in the following sections.

7.3.1 Improvement of Test Data Search in the Presence of Nesting and Short-Circuiting Logical Operators

This is a general problem for evolutionary structural test data generation, which was raised in Section 2.3.5. The problem is accentuated for finding test data to execute event sequences, since there are more nodes to execute in turn. If the test data search fails to find inputs to execute just one of the many nodes in the sequence, the search for the overall event sequence will be deemed a failure.

Baresel *et al.* propose a solution where all nested conditions are evaluated at the same time [BSS02] (described in Section 2.3.5). However, all conditions can only be evaluated at once if the variables involved are not modified between one condition and the next. As suggested in Section 2.3.5, a side-effect removal transformation [HHZM01] or the use of temporary variables could overcome side-effects in conditions composed using short-circuiting logical operators. For nested conditions, variables should not be modified between the outer condition and the inner nested condition. In the following code `b` is modified between the `if` statements, and so the condition `(b == c)` cannot be evaluated at the same time as `(a == b)`:

```
if (a == b)
    b ++;
    if (b == c)
        // execute this statement
```

However, a testability transformation approach [HHH+04], using an extension of the temporary variable idea suggested for composed conditions, could be applied in this instance. The value of `b` could be assigned to a temporary variable before the first condition and then incremented, which the latter condition evaluated using the temporary variable:

```
t = b;
t ++;
  – evaluate (a == b ∧ t == c) here for objective function
if (a == b)
    b ++;
    if (b == c)
        // execute this statement
```

The conditions can now be evaluated for the purposes of computing the objective function before the nested `if` structure. The approach becomes more complicated when the variables involved in nested conditions are modified within nested structures themselves, for example:

```
if (a == b)
    if (a == 0)
        b ++;
    else
        b --;
    if (b == c)
        // execute this statement
```

However this could be transformed as follows:

```
t1 = b;
t1 ++;
t2 = b;
t2 --;
  – evaluate $(a == b) \wedge ((a == 0 \wedge t1 == c) \vee (\neg a == 0 \wedge t2 == c))$
  here for objective function
if (a == b)
    if (a == 0)
        b ++;
    else
        b --;
    if (b == c)
        // execute this statement
```

where the objective function is modified so that both possible values for `b` when the nested condition is reached can be taken into account.

Care needs to be taken when transforming conditions involving array references or pointers, due to the possibility of introducing array out of bounds or null pointer errors.

7.3.2 Static Analysis to Reduce the Size of the Chaining Tree

Further static analysis could be used to rule out unhelpful event sequences and limit the growth of the chaining tree. For example, flag variables are often assigned constant true or false values. If the current problem node requires a flag to be true, then all last definitions involving the flag being set to false can be safely ignored. However, the current algorithm for finding last definitions does not analyse values assigned to variables.

Further rules could be included for the increment (`++`) and decrement (`--`) operators. Suppose a problem node requires some variable `counter` to be equal to 3, and the current value of `counter` is less than 3, then all last definitions involving the decrement operator can be safely ignored.

7.3.3 Search Space Reduction using Variable Dependence Analysis

The VADA tool mentioned in Section 2.3.5 can reduce the search space for finding test data by only considering the ranges of input variables that actually have an outcome on the test goal.

CFG Node	
s	`void counter_example(int a)`
	`{`
1	`    static int counter = 0;`
2	`    if (a == 0)`
3	`        counter ++;`
4	`    if (counter == 5)`
5	`        // target statement`
	`}`

Figure 7.2: Example for future research with search space reduction

Similar analysis could be applied to the finding of test data for event sequences with the state problem, since the input domain can become very large as soon as a sequence of function calls is required.

Furthermore, test data could be reused from one execution of a function to the next. In the example of Figure 7.2 the `counter` variable needs to be incremented five times in order for the target statement to be executed. The increment of the counter is purely dependent on a condition relating to the inputs of the current function call (i.e. not on the values of other internal storage variables). Therefore, the search only needs to find test data to execute this increment statement once, and repeat this input for the other four calls, rather than the search having to find five calls where the input value is zero on each occasion.

7.3.4 Analysis of Assignments as a Result of Boolean Conditions

The current method does not appreciate that variables assigned as the result of boolean conditions, for example:

```
flag = (a > b);
```

have two distinct possible outcomes. This information could provide extra guidance to the search. Currently the method cannot try and force `flag` to be either true or false. The assignment is in fact equivalent to the following code:

```
if (a > b)
```

```
        flag = true;
    else
        flag = false;
```

In this instance, if the hybrid approach wanted to force one of the assignment outcomes, the `if` statement would become a problem node, and the condition `a > b` would be used to guide the search to one of the assignments. Further work needs to extend the method so that boolean assignments are treated in a similar manner to the latter version involving the `if` statement.

A similar situation occurs with the use of the ternary operator, for example:

```
    x = (a > b) ? -1 : 1;
```

Again, the method does not recognize there are two distinct assignments possible to `x`. This statement is actually equivalent to the following code:

```
    if (a > b)
        x = -1;
    else
        x = 1;
```

which can be handled with ease by the hybrid approach.

7.3.5 Possible Improvement of Efficiency via Seeding

The best test data found for event sequences could be used to seed the search for child event sequences. Take the example of Figure 7.3. The initial event sequence is:

$$< (s, \emptyset), (6, \emptyset) >$$

The best test data diverges down the false branch at node 5 - where `flag` needs to be true. The test data vector has the input `b` as zero in order to execute node 4 as true. This input vector could be used to seed the searches of the child event sequences, since `b` will still be required to be zero in order for node 6 to be reached.

Future work needs to investigate whether seeding test data searches for child event sequences will in general improve efficiency, and reduce the number of objective function evaluations needed to find test data; since such a strategy

```
CFG
Node
 s    void seeding_example(int a, int b)
      {
 1        int flag = 0;
 2        if (a == 0)
 3            flag = 1;

 4        if (b == 0)
          {
 5            if (flag)
 6                // target statement
          }
 e    }
```

Figure 7.3: Example for future research with seeding

could also disadvantage the search by initializing it at an unhelpful starting point, leading to more problem nodes being encountered.

7.3.6 Possible Improvement of Test Data Search using Multi-Objective Optimization

The objective function for analyzing test data with respect to an event sequence simply adds up the individual objective values of each event. Take the example of Figure 7.4 and the event sequence:

$$< (s, \emptyset), (3, \{flag1\}), (5, \{flag1, flag2\}), (6, \emptyset), (7, \emptyset) >$$

consider the two input vectors:

(1) `(a=0, b=1)`

(2) `(a=1, b=0)`

For input vector 1, the objective value of the second event, $e_2 = (3, \{flag1\})$ is zero because the event is successfully executed, yet the objective value of the third event $e_3 = (5, \{flag1, flag2\})$ has a branch distance of 1 since `b=1` and node 5 is missed. The objective value of the input vector is the normalized branch distance value of 1. For input vector 2, the reverse is true. The objective value of the second event e_2 is the normalized branch distance value of 1 since node 3 is missed, yet the objective value of the third event is zero.

```
CFG
Node
 s    void multiobjective_opt_example(int a, int b)
      {
 1        static int flag1 = 0, flag2 = 0;

 2        if (a == 0)
 3            flag1 = 1;

 4        if (b == 0)
 5            flag2 = 1;

 6         if (flag1 && flag2)
 7            // target statement
      }
```

Figure 7.4: Example for future research with multi-objective optimization

Despite the fact that these different input vectors have different merits, they both have the same overall objective value. Instead of cumulating the objective values for each event, multi-objective optimization [CC99, Deb00] could be used to optimize each event individually.

7.3.7 Heuristic Chaining Tree Search

If none of the static analysis techniques outlined above can limit the chaining tree size, then it may be possible to search the chaining tree heuristically. Simple initial experiments have been carried out by McMinn and Holcombe [MH03] with the use of Ant Colony Optimization.

Appendix A

Experimental Framework

A.1 Introduction

"ETState" is the name given to the experimental framework written to perform all the experiments in this thesis. It is coded in Java, using version 1.4.2 of the standard developer's kit. It is designed to run in the Cygwin environment on the Microsoft Windows platform. Cygwin is a Linux-style environment for Windows operating systems [cyg04].

The framework implements the standard evolutionary approach (Section 3.3.2), the sequence evolutionary approach (Section 3.5), random testing (Sections 3.3.2 and 3.6.1), the hybrid approach (incorporating the extended chaining approach) (Section 5.2), and the sequence hybrid approach (Section 6.2).

The framework ran on a Pentium IV PC running at 3GHz with 1Gb of RAM, under the Windows XP operating system under normal load conditions. The Cygwin environment used was version 1.5.9-1.

Implementation details of interest discussed here but not elsewhere in this thesis include the preparation of test objects for experimental use, including parsing, instrumentation and compilation; and details of the implementation of the test data search, including interaction with the evolutionary algorithm, execution of the test object, and collation of objective function values.

A.1.1 Overview of the System

Figure A.1 shows the components of the experimental framework. The test object is prepared, as outlined in Section A.2, and compiled to a dynamic link library (DLL). The DLL is compiled with the Java Native Interface (JNI) headers so that it can be called directly by the ETState Java code.

The Peanuts evolutionary algorithm server was kindly provided by Daimler-

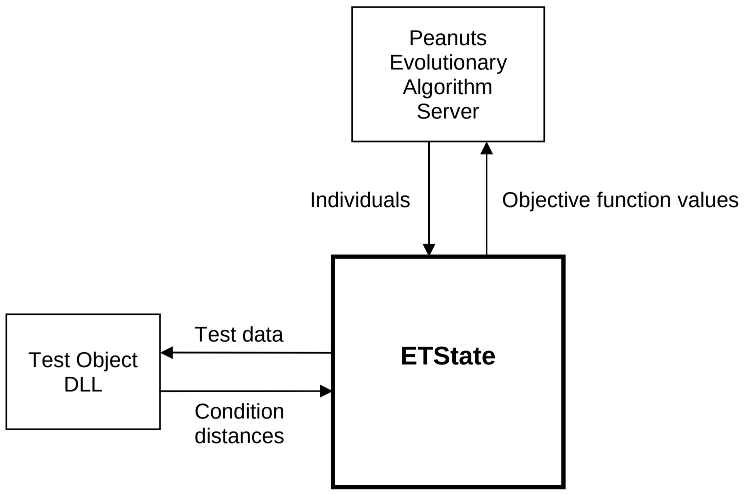

Peanuts Evolutionary Algorithm Server
Individuals
Objective function values
Test Object DLL
Test data
Condition distances
ETState

Figure A.1: Overview of ETState test data evaluation process

Chrysler Research and Technology, and provides an interface to the publically available GEATbx (Genetic and Evolutionary Algorithm Toolbox) MATLAB library [gea04], written by Harmut Polheim. The Peanuts server runs on the Windows platform, communicating over a network connection through a socket, and uses GEATbx version 3.3. For all the experiments, the Peanuts server was run on the same machine as ETState.

The Peanuts server sends individuals to ETState, which executes the test object with the decoded test data. The test object is instrumented so that ETState can be reported of the "condition distances" of conditions encountered during execution of the program. This is described in detail in Sections A.2.4, A.3.2 and A.3.3. From this information, ETState works out the path executed and the overall objective value for the individual. Different objective functions are used for different purposes in this thesis. Objective values are sent to the Peanuts server, which uses this information to recombine and mutate the population. A new generation of individuals is produced and sent to ETState. The cycle continues until the required test data is found or the termination criterion being used is met.

A.2 Preparation of Test Objects

Each test object is prepared according to the following steps:

1. Inlining of embedded function calls

2. Parsing of source code to XML

3. Creation of internal model of source code
4. Test object instrumentation
5. Addition of reset functionality
6. Compilation of the modified test object to a dynamic link library

A.2.1 Inlining of Embedded Function Calls

The implementation of the extended chaining approach cannot currently analyse data dependencies within embedded function calls, so for the purposes of this thesis, the code contained within these calls is directly inlined into the calling function by hand. The tools could easily be extended in future so that embedded function calls can be analyzed, and this stage subsequently removed. If the function call takes place within a branch condition, the function is inlined immediately before the condition, and the return value is assigned to a temporary variable. This temporary variable then replaces the function call in the condition. Functions passing and returning values through pointer arguments cannot currently be handled.

A.2.2 Parsing of Source Code to XML

Using a C parser kindly provided by DaimlerChrysler, the source code of the test object is parsed and an abstract syntax tree is produced in XML format. An interface specification is also created in XML format using a separate tool also provided by DaimlerChrysler. The interface specification consists of the ranges and precision of each input variable of each function in the test object.

A.2.3 Creation of Internal Model of the Test Object Source Code

The abstract syntax tree of the code is read in, and an internal model of the code is created by ETState. This includes a control flow and control dependence graph. The control dependence graph is used in the calculation of approach levels in assigning objective function values (as described in Section 2.3.5). The control flow graph is used to find variable definitions when creating event sequences. Information about the composition of atomic conditions within decisions is also collated, as this is information for computing branch distance values (see Section A.3.3).

A.2.4 Test Object Instrumentation

Each atomic condition appearing in a decision expression in the test object is replaced by a function which takes two or more parameters - the identification number of the condition, followed by the values of the variables or constants involved in the condition. These functions compute the true and false "condition distances" using the values used at the condition. The condition distances are computed using the rules in Table 2.2. The value of K used is the smallest level of precision for the types of the variables used in the condition. For example, if a condition involves the comparison of two integer variables the value of K is 1.

The condition distances and the condition identification number is fed to ETState for objective function computation purposes. ETState computes the complete true and false branch distances using the individual atomic conditions (see Section A.3.3).

The boolean outcome of the original condition is returned by the function so that the normal flow of execution through the program is unchanged.
For example the following `if` statement:

```
if ((i1 > i2) && (d1 < d2))
{
   // ...
```

would be instrumented as follows:

```
if (grt_than_int(5, i1, i2)) && less_than_dbl(6, d1, d2))
{
   // ...
```

In the example, the function `grt_than_int` sends the true and false atomic condition distances for predicates of the form $a \geq b$ for variables of type `integer` to ETState. The function `less_than_dbl` sends the true and false condition distances of predicates of the form $a \leq b$ for variables of type `double` to ETState.

Instrumentation of test objects was performed manually, but could be automated by creation of further tools.

A.2.5 Addition of Reset Functionality

In order to generate test sequences, is it necessary to be able to reset the test objects to its initial state as required. This involves the addition of some extra functionality by hand, if it is not already present. Again, the stages outlined in this section could be performed automatically by the creation of further tools.

Resetting Single Function Test Objects

In order to reset single function test objects with internal data declared using the C `static` storage class, two manual changes are made to the test object so that it can be reset in the course of testing.

The first change is the addition of an extra flag input to the interface of the function. When the input value of the flag is true, the test object is reset. The second change is the addition of a block of code to reset the internal storage variables when the flag is true. This block is inserted into the function immediately after the declaration of the static variables. After the reset is performed, a `return` instruction is added to terminate execution of the function.

For example, the following function:

```
void reset_example(int in1, int in2)
{
    static int internal_var = 0;

    if (in1 > in2)
        internal_var ++;

    // ...
}
```

would be modified as follows:

```
void reset_example(int in1, int in2, int reset)
{
    static int internal_var = 0;

    if (reset)
    {
        internal_var = 0;
        return;
    }

    if (in1 > in2)
        internal_var ++;

    // ...
}
```

A.2.6 Resetting Multiple Function Test Objects

In order to reset modules with multiple functions, an extra reset function was added to reset the internal state variables global to each function.

For example, the following module would have the following reset function added:

```
static int counter = 0;

void function1(int a)
{
    if (a > 0)
        counter ++;
}

//...

void reset()
{
    counter = 0;
}
```

A.2.7 Compilation of Test Object to a Dynamic Link Library

The test object, functions computing condition distances, and Java Native Interface (JNI) headers are compiled to produce a dynamic link library (DLL) so that the test object can be directly called by the ETState code which is written in Java.

Compilation is performed using the GCC C compiler version 3.3.1 for the Cygwin environment.

A.3 Implementation Details of the Test Data Search

The test data search is a cycle involving the following steps

1. Retrieving a generation of individuals from the evolutionary algorithm
2. Decoding individuals to test data inputs
3. Calling of the test object with the test data
4. Tracing the flow of execution through the test object, retrieving branch distance values

5. Assigning objective function values to individuals.

A.3.1 The Evolutionary Algorithm Toolbox

The Peanuts server optimizes a vector of variables of integer or floating point type. The first stage of communication involves sending a specification of the vector of variables to be optimized, namely the maximum and minimum value of each variable, and the precision to be used. The Peanuts server then sends the first generation of randomly generated individuals. The client (ETState) then performs objective function evaluations, and sends these values backs to the Peanuts server, which recombines and mutates the individuals and sends the next generation. The search terminates when test data has been found (an objective value of zero has been found) or the stopping criterion (e.g. a maximum number of generations, or a maximum number of generations without an improvement in the best objective value) has been met.

An options file used by the server sets the number of individuals in a generation, the number of sub-populations, stopping criteria and so on (see Section A.4).

A.3.2 Decoding Individuals, Test Object Execution, and Condition Distance Feedback

The variable values making up each individual are decoded to test object inputs where necessary (i.e. for the multi-function call sequence encoding described in Sections 3.5 and 6.2, as part of the sequence approaches). The test object DLL is then called with these input values. Condition distances are sent to ETState corresponding to the atomic conditions evaluated during test object execution, as a result of the instrumentation described in Section A.2.4. Using this information, the path through the test object can be traced.

Take the example of Figure A.2. Suppose the test object is executed with the input vector `(a=10, b=20, c=30)`. For each condition and corresponding true or false outcome, a numerical distance is returned to ETState (this is because the condition may need to be evaluated to either outcome, depending on the event sequence being considered):

Condition	Outcome	Predicate	Numerical Distance
1	True	$a < b$	0
1	False	$a \geq b$	$20 - 10 + K = 21$
2	True	$a > b$	$20 - 10 + K = 21$
2	False	$a \leq b$	0

```
 CFG
 Node
  s    void test_object(int a, int b, int c)
       {
  1        if (less_than_int(1, a, b))
           {
  2            // perform some action
           }

  3        if (grt_than_int(2, a, b) && grt_than_int(3, b, c))
           {
  4            if (eql_to_int(4, a, c))
               {
  5                // perform some action
               }

  6            // perform some action
           }
  e    }
```

Figure A.2: Example for tracing execution paths

Numerical distances are computed using the rules in Table 2.2 where $K = 1$. Condition distances of zero indicate the condition was evaluated to the corresponding outcome. Therefore it can be seen that the true branch was taken from node 1 (as condition 1 evaluated to true), followed by the false branch from node 3 (as condition 2 evaluated to false). Condition 3 was never evaluated because evaluation of the overall decision at node 3 short-circuited after condition 2, which had already determined the decision to be false. Node 4 was never reached, so condition 4 was not evaluated either. The path taken was therefore $< s, 1, 2, 3, e >$.

Approach levels for program structures can be found by using the generated control dependence graph and by analyzing the path of execution taken through the test object, as described in Section 2.3.5.

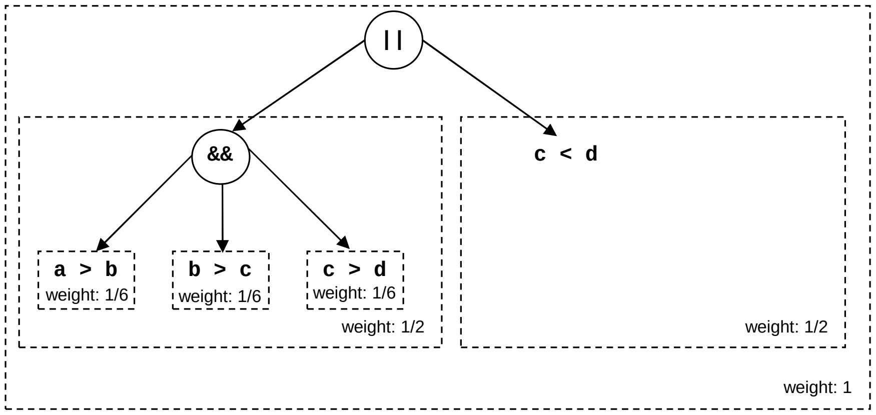
A.3.3 Computing Branch Distance Values

Branch distances are computed by ETState using the individual condition distances fed back from test object execution. Where predicates consist of one condition, the true and false branch distances are equivalent to the respective true and false condition distances, normalized into the range 0-1 using Equation 2.1.

However, when the branch predicate is made up of several conditions using logical connectives, the branch distance must be assembled from the individual conditions. The usual procedure for doing this is to employ Tracey's rules for logical connectives (Table 2.3). However with the C language, some conditions may not be evaluated due to the use of short circuit operators. The method used by ETState is to weight the distances of each atomic condition in the overall composed condition, using assumed values for unevaluated conditions. A similar algorithm is applied by Baresel [Bar02], but does not appear in the literature.

In order to assign weights to each atomic condition, the overall expression must first be decomposed into a tree, where the leaves of the tree are the atomic conditions and the nodes are the short-circuit operators.

For example the expression `((a > b && b > c && c > d) || (c < d))` is decomposed to the following tree:



Starting at the root of the tree with an initial weight of 1, the weight of a subcondition is assigned by dividing the weight of its parent condition across itself and its siblings.

In the example, the weight of the overall expression is 1. Its subconditions `(a > b && b > c && c > d)` and `(c < d)` stemming from the `||` operator are each assigned a weight of $\frac{1}{2}$. The individual atomic conditions of `a > b`, `b > c`, `c > d` are each assigned a weight of $\frac{1}{2} \div 3 = \frac{1}{6}$.

The next step is to normalize each individual atomic condition distance into the range 0-1 using Equation 2.1. If a condition is unevaluated, the normalized distance is assumed to be 0 or 1 depending on whether its parent condition was evaluated as true or false. For example if `b > c` is unevaluated and `(a > b && b > c && c > d)` is false, its true normalized distance is 1, and its false normalized distance is 0. Each normalized atomic condition distance is multiplied by the weight assigned to that atomic condition.

Finally, Tracey's rules for logical connectives (Table 2.3) are applied to find the overall branch distance. For the example, the true distance of the expression is calculated as:

$$min((\frac{norm\_true\_dist(a>b)}{6} + \frac{norm\_true\_dist(b>c)}{6} + \frac{norm\_true\_dist(c>d)}{6}), \frac{norm\_true\_dist(c<d)}{2})$$

where $norm\_true\_dist$ is the normalized true condition distance of the condition.

A.4 Evolutionary Search Parameters

Evolutionary searches were conducted using version 3.3 of the publically available GEATbx library [gea04]. The GEATbx options file used is shown in Figure A.3. Apart from the termination criterion, the same set of search parameters were applied to each evolutionary search undertaken across all experiments performed in this thesis. This parameter set is the same as that applied by other authors in the field, for example Harman *et al.* [HHH+02] and Baresel *et al.* [BPS03]. Individuals are composed of a vector of variables of integer and floating-point types. The range is specified for each variable, as well as the precision for floating-point variables. 300 individuals were used per generation. This is split into 6 subpopulations of approximately 50 individuals each. Linear ranking is utilized, with a selection pressure of 1.7. Individuals are recombined using discrete recombination, and mutated using real-valued mutation.

Real-valued mutation is performed using "number creep" - the mutation of variable values through the addition of randomly created values. The probability of mutating a variable value is the inverse of the number of variables in the individual. The formula for mutating a variable value var_i where i is the locus of the variable in the chromosome is that as used by the "Breeder Genetic Algorithm" [MSV93], is stated as follows:

$$var_i^{mut} = var_i + s_i \times r_i \times a_i$$

where:

- $s_i \in \{-1, 1\}$ chosen randomly
- $r_i = r \times domain_i$, where r is the "mutation range" and $domain_i$ is the size of the domain of the variable i
- $a_i = 2^{-u \times k}$, where u is a random number between 0 and 1, and k is the "mutation precision".

The smallest relative mutation step size is 2^{-k} with the largest being $2^0 = 1$. Mutation steps are therefore created in the range $[r_i \times 2^{-k}, r_i]$. The mutation precision value k used was 16. Each subpopulation was assigned a different value of r, this being 10^{-n}, where n is the subpopulation number, $1 \leq n \leq 6$.

Individuals migrate from one subpopulation to another throughout the progression of the search. The top 10% of each subpopulation are copied to the migration pool. From this pool, the individuals are randomly selected to be reinserted back into each subpopulation, replacing the worst individuals. Copying of individuals back into the same subpopulation is prevented by restricting the random selection to individuals from other subpopulations only. Subpopulations compete with one another, and so the number of individuals copied back depends on the success of the population. This is determined by ranking each subpopulation according to the average objective value of the individuals within it. Unsuccessful subpopulations are not allowed to die off completely, with the minimum number of individuals in a subpopulation being 5.

```
% Specify 6 subpopulations of 50 individuals each:
NumberSubpopulation:  6
NumberIndividuals:  50

% Specify discrete recombination:
Recombination.Name:  'recdis'

% Specify real-valued mutation, with following ranges and precision:
Mutation.Name:  'mutreal'
Mutation.Range:  0.1 0.01 0.001 0.0001 0.00001 0.000001
Mutation.Precision:  16

% Terminate on maximum number of generations or found objective
% function value of 0:
Termination.Method:  1 3

% Termination.MaxGenerations varies for different experimental setups:
Termination.MaxGenerations:  200

% Terminate only when reach zero value of objective function:
Termination.Diff2Optimum:  0

% Known minimum of the objective function (best objective value):
System.ObjFunMinimum:  0

% Do only one minimization:
System.ObjFunAddPara:  1
```

Figure A.3: GEATbx option file

Appendix B

Program Code for Synthetic Test Objects with State Behaviour

This section archives the program code for synthetic test objects with state behaviour, as used in the experimental studies of Chapter 3 and Chapter 6. Line numbers appear to the left of each line of code.

B.1 Anomaly Detector

```
#define MAX_ELEMENTS 40

static double vibration[MAX_ELEMENTS];
static int buffer_full = 0;
static int ptr = 0;

void add_data(double current_vibration)
{
    vibration[ptr] = current_vibration;
    ptr ++;

    if (ptr == MAX_ELEMENTS)
    {
        ptr = 0;
        buffer_full = 1;
    }
}

int normal_limits()
{
    if (buffer_full)
    {
        double total = 0, mean = 0,
               variance = 0, last_element = 0;
        int i, j;
        int cur_pos = ptr-1;

        if (cur_pos == -1)
        {
            cur_pos = MAX_ELEMENTS-1;
        }

        for (i=0; i < MAX_ELEMENTS; i++)
        {
            if (i != cur_pos)
            {
                total += vibration[i];
            }
        }

        mean = total / (MAX_ELEMENTS-1);
        total = 0;

        for (j=0; j < MAX_ELEMENTS; j++)
        {
            if (j != cur_pos)
            {
                total += (vibration[j] - mean) *
                         (vibration[j] - mean);
            }
        }

        variance = total / MAX_ELEMENTS-1;
        last_element = (vibration[cur_pos] - mean) *
                       (vibration[cur_pos] - mean);

        return (last_element > variance);
    }
    else
    {
        return 1;
    }
}

void reset()
{
    buffer_full = 0;
    ptr = 0;
}
```

B.2 Array Difference

```
#define SIZE 10
typedef unsigned char Bool;

Bool diff(int incoming[SIZE])
{
    static int last[SIZE];
    static Bool init = 1;

    int diff = 0;
    int i;

    if (!init)
    {
        for (i=0; i < SIZE; i++)
        {
            if (incoming[i] != last[i])
            {
                diff = 1;
            }
        }
    }

    if (diff || init)
    {
        for (i=0; i < SIZE; i++)
        {
            last[i] = incoming[i];
        }
    }

    if (init)
    {
        init = 0;
    }

    return diff;
}
```

B.3 Postcode

```
#define RESULT_ENTER_NEW_CHAR 0
#define RESULT_VALID 1
#define RESULT_INVALID 2

typedef unsigned char Bool;

static Bool is_letter(int unicode_char)
{
    if (unicode_char >= 65 && unicode_char <= 90)
        return 1;
    else
        return 0;
}

static Bool is_digit(int unicode_char)
{
    if (unicode_char >= 48 && unicode_char <= 57)
        return 1;
    else
        return 0;
}

static Bool is_space(int unicode_char)
{
    if (unicode_char == 32)
        return 1;
    else
        return 0;
}

static Bool is_lf(int unicode_char)
{
    if (unicode_char == 12)
        return 1;
    else
        return 0;
}

int validate_uk_postcode(int unicode_char)
{
    static Bool pri_letter_1 = 1;
    static Bool pri_letter_2 = 0;
    static Bool pri_digit_1  = 0;
    static Bool pri_digit_2  = 0;
    static Bool space        = 0;
    static Bool sec_digit    = 0;
    static Bool sec_letter_1 = 0;
    static Bool sec_letter_2 = 0;
    static Bool end          = 0;

    int result = RESULT_ENTER_NEW_CHAR;

    if (pri_letter_1)
    {
        if (!is_letter(unicode_char))
        {
            result = RESULT_INVALID;
        }
        else
        {
            pri_letter_2 = 1;
            pri_letter_1 = 0;
        }
    }
    else if (pri_letter_2)
    {
        if (is_letter(unicode_char))
        {
            pri_digit_1 = 1;
            pri_letter_2 = 0;
        }
        else if (is_digit(unicode_char))
        {
            pri_digit_2 = 1;
            pri_letter_2 = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (pri_digit_1)
    {
        if (is_digit(unicode_char))
        {
            pri_digit_2 = 1;
            pri_digit_1 = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (pri_digit_2)
    {
        if (is_digit(unicode_char))
        {
            space = 1;
            pri_digit_2 = 0;
        }
        else if (is_space(unicode_char))
        {
            sec_digit = 1;
            pri_digit_2 = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (space)
    {
        if (is_space(unicode_char))
        {
            sec_digit = 1;
            space = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (sec_digit)
    {
        if (is_digit(unicode_char))
        {
            sec_letter_1 = 1;
            sec_digit = 0;
        }
        else if (is_letter(unicode_char))
        {
            sec_letter_2 = 1;
            sec_digit = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (sec_letter_1)
    {
        if (is_letter(unicode_char))
        {
            sec_letter_2 = 1;
            sec_letter_1 = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (sec_letter_2)
    {
        if (is_letter(unicode_char))
        {
            end = 1;
            sec_letter_2 = 0;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (end)
    {
        if (is_lf(unicode_char))
        {
            pri_letter_1 = 1;
            end = 0;
            result = RESULT_VALID;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }

    return result;
}
```

B.4 Sliding Window

```
#define SEND_WINDOW_SIZE 3
#define RECEIVE_WINDOW_SIZE 3
#define MSG_SIZE 20

typedef unsigned char Bool;

// SENDER SIDE STATE
static int sender_seq_no = 0;
static int sender_last_ack_received = -1;
static int sender_last_frame_sent = -1;

static int sent[SEND_WINDOW_SIZE];
static int sender_data[SEND_WINDOW_SIZE][MSG_SIZE];

// RECEIVER SIDE STATE
static int receiver_next_frame_expected = 0;

static int received[RECEIVE_WINDOW_SIZE];
static int received_data[RECEIVE_WINDOW_SIZE][MSG_SIZE];

static Bool send_window_not_full()
{
    if (sender_last_frame_sent - sender_last_ack_received
            < SEND_WINDOW_SIZE)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

static Bool is_in_window(int seq_no, int min, int max)
{
    int pos = seq_no - min;
    int max_pos = max - min + 1;
    if (pos < max_pos)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

static Bool is_next_frame(int seq_no, int next_frame)
{
    if (seq_no == next_frame)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

static send_msg_on_network(int seq_no, int data[MSG_SIZE])
{
    // not implemented
}

static send_ack_on_network(int seq_no)
{
    // not implemented
}

static deliver_to_application(int data[MSG_SIZE])
{
    // not implemented
}

void initialise()
{
    int i;
    for (i=0; i < SEND_WINDOW_SIZE; i++)
    {
        sent[i] = 0;
    }
    for (i=0; i < RECEIVE_WINDOW_SIZE; i++)
    {
        received[i] = 0;
    }

    sender_seq_no = 0;
    sender_last_ack_received = -1;
    sender_last_frame_sent = -1;
    receiver_next_frame_expected = 0;
}
```

```
Bool send(int msg[MSG_SIZE])
{
    if (send_window_not_full())
    {
        int slot = sender_seq_no % SEND_WINDOW_SIZE;
        sent[slot] = 1;
        sender_last_frame_sent ++;
        sender_seq_no ++;
        send_msg_on_network(sender_seq_no, msg);
        return 1;
    }
    else
    {
        return 0;
    }
}

void receive_ack(int ack_no)
{
    // sender side
    if (is_in_window(ack_no,
                     sender_last_ack_received + 1,
                     sender_last_frame_sent))
     {
        while (sender_last_ack_received != ack_no)
        {
            int slot = (sender_last_ack_received + 1) % SEND_WINDOW_SIZE;
            sent[slot] = 0;
            sender_last_ack_received ++;
        }
     }
}

void receive_frame(int seq_no, int data[MSG_SIZE])
{
    if (is_in_window(seq_no,
                     receiver_next_frame_expected,
                     receiver_next_frame_expected + RECEIVE_WINDOW_SIZE - 1))
    {
        int slot = seq_no % RECEIVE_WINDOW_SIZE;
        int i;

        received[slot] = 1;
        for (i=0; i < MSG_SIZE; i++)
        {
            received_data[slot][i] = data[i];
        }
        if (is_next_frame(seq_no, receiver_next_frame_expected))
        {
            while (received[slot])
            {
                deliver_to_application(received_data[slot]);

                received[slot] = 0;
                receiver_next_frame_expected ++;
                slot = receiver_next_frame_expected % RECEIVE_WINDOW_SIZE;

                receiver_next_frame_expected ++;
            }
            send_ack_on_network(seq_no);
        }
    }
    // else ignore frame
}
```

B.5 Smoke Detector

```
const double LEVEL = 0.003;
const int DANGER = 3;
const int WAIT_TIME = 3;

void smoke_detector(double level, int* signal_on, int* signal_off)
{
    static int time = 0, off_time = 0;
    static int detected = 0, alarm_on = 0, waiting = 0;

    time ++;

    if (level > LEVEL && detected < DANGER)
    {
        detected ++;
    }
    else if (level <= LEVEL && detected > 0)
    {
        detected --;
    }

    if (!alarm_on && detected == DANGER)
    {
        alarm_on = 1;
        *signal_on = 1;
    }

    if (alarm_on)
    {
        if (!waiting && detected == 0)
        {
            waiting = 1;
            off_time = time + WAIT_TIME;
        }

        if (waiting && detected == DANGER)
        {
            waiting = 0;
        }

        if (waiting && time >= off_time)
        {
            waiting = 0;
            alarm_on = 0;
            *signal_off = 1;
        }
    }
}
```

B.6 Sortcode

```
#define RESULT_ENTER_NEW_CHAR 0
#define RESULT_VALID 1
#define RESULT_INVALID 2

typedef unsigned char Bool;

static Bool is_digit(int unicode_char)
{
    if (unicode_char >= 48 && unicode_char <= 57)
        return 1;
    else
        return 0;
}

static Bool is_dash(int unicode_char)
{
    if (unicode_char == 45)
        return 1;
    else
        return 0;
}

static Bool is_lf(int unicode_char)
{
    if (unicode_char == 12)
        return 1;
    else
        return 0;
}

int validate_sortcode(int unicode_char)
{
    static Bool digit_set_1 = 1;
    static Bool digit_set_2 = 0;
    static Bool digit_set_3 = 0;
    static Bool dash_1      = 0;
    static Bool dash_2      = 0;
    static Bool end          = 0;
    static int  pos         = 0;

    int result = RESULT_ENTER_NEW_CHAR;

    if (is_digit(unicode_char))
    {
        if (digit_set_1 && pos == 1)
        {
            digit_set_1 = 0;
            dash_1      = 1;
        }
        else if (digit_set_2 && pos == 4)
        {
            digit_set_2 = 0;
            dash_2      = 1;
        }
        else if (digit_set_3 && pos == 7)
        {
            digit_set_3 = 0;
            end         = 1;
        }

        if (pos != 2 && pos != 5)
        {
            pos ++;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (is_dash(unicode_char))
    {
        if (dash_1)
        {
            dash_1      = 0;
            digit_set_2 = 1;
        }
        else if (dash_2)
        {
            dash_2      = 0;
            digit_set_3 = 1;
        }

        if (pos == 2 || pos == 5)
        {
            pos ++;
        }
        else
        {
            result = RESULT_INVALID;
        }
    }
    else if (is_lf(unicode_char) && end && pos == 8)
    {
        result = RESULT_VALID;
        digit_set_1 = 1;
        end = 0;
        pos = 0;
    }
    else
    {
        result = RESULT_INVALID;
    }

    return result;
}
```

B.7 Stack

```
#define MAX_ELEMENTS 40
#define NO_ERROR 0
#define STACK_UNDERFLOW 1
#define STACK_OVERFLOW 2

static double elements[MAX_ELEMENTS];
static int size = 0;
static int error = 0;

double pop()
{
    error = NO_ERROR;

    if (size <= 0)
    {
        error = STACK_UNDERFLOW;
    }

    if (error == NO_ERROR)
    {
        size --;
        return elements[size];
    }
    else
    {
        return 0;
    }
}

void push(double element)
{
    error = NO_ERROR;

    if (size > MAX_ELEMENTS)
    {
        error = STACK_OVERFLOW;
    }

    if (error == NO_ERROR)
    {
        elements[size] = element;
        size ++;
    }
}
```

```
int check_error()
{
    return error;
}

int check_size()
{
    return size;
}

void reset()
{
    size = 0;
    error = 0;
}
```

B.8 Telephone Number

```
#define LINE_FEED_CHAR 12
#define ZERO_CHAR 48
#define ONE_CHAR 49
#define TWO_CHAR 50
#define THREE_CHAR 51
#define FOUR_CHAR 52
#define FIVE_CHAR 53
#define SIX_CHAR 54
#define SEVEN_CHAR 55
#define EIGHT_CHAR 56
#define NINE_CHAR 57

typedef unsigned char Bool;

void validate_uk_tel_no(int unicode_char, Bool* valid, Bool* error)
{
    static int position = 0;
    static Bool local = 0;
    static Bool international = 0;
    static Bool national = 0;

    *valid = 0;
    *error = 0;

    if (unicode_char == LINE_FEED_CHAR)
    {
           if (local && position == 7)
           {
               *valid = 1;
           }
           else if (national && position == 11)
           {
               *valid = 1;
           }
           else if (international && position == 14)
           {
               *valid = 1;
           }
           else
           {
               *error = 1;
           }
       }
       else if (unicode_char >= ZERO_CHAR && unicode_char <= NINE_CHAR)
    {
        if (position == 0)
        {
            if (unicode_char == ZERO_CHAR)
               {
                   national = 1;
               }
               else
               {
                local = 1;
               }
           }

           if (position == 1 && unicode_char == ZERO_CHAR && national)
           {
               international = 1;
               national = 0;
           }

           if ((position == 2 || position == 3) &&
                   unicode_char != FOUR_CHAR && international)
           {
            *error = 1;
           }

        if (position == 4 && international && unicode_char == ZERO_CHAR)
        {
            *error = 1;
        }

           position ++;
       }
       else
       {
           *error = 1;
       }

       if (*error)
       {
        position = 0;
        international = 0;
        local = 0;
        national = 0;
       }
}
```

B.9 Vending Machine

```
typedef unsigned char Bool;

#define CHOC_COST 9
#define CRISPS_COST 8
#define COKE_COST 6

// change
static int ten_pennies = 10;
static int five_pennies = 10;
static int two_pennies = 10;
static int one_pennies = 10;

// stock of various items
static int no_of_choc = 10;
static int no_of_crisps = 10;
static int no_of_coke = 10;

// initialise
static int inserted = 0;

void reset()
{
    ten_pennies = 10;
    five_pennies = 10;
    two_pennies = 10;
    one_pennies = 10;
    no_of_choc = 10;
    no_of_crisps = 10;
    no_of_coke = 10;
    inserted = 0;
}

static Bool check_have_change()
{
    if (ten_pennies > 0 && five_pennies > 0 &&
            two_pennies > 0 && one_pennies > 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void insert_coinage(int coin, int amount)
{
    if (coin == 1)
    {
        one_pennies += amount;
        inserted += amount;
    }
    else if (coin == 2)
    {
        two_pennies += amount;
        inserted += amount * 2;
    }
    else if (coin == 5)
    {
        five_pennies += amount;
        inserted += amount * 5;
    }
    else if (coin == 10)
    {
        ten_pennies += amount;
        inserted += amount * 10;
    }
}

void buy_item(int item_id)
{
    if (item_id == 0 && no_of_choc > 0)
    {
        if (inserted > CHOC_COST)
        {
            inserted -= CHOC_COST;
            no_of_choc -= 1;
        }
    }
    else if (item_id == 1 && no_of_crisps > 0)
    {
        if (inserted > CRISPS_COST)
        {
            inserted -= CRISPS_COST;
            no_of_crisps -= 1;
        }
    }
    else if (item_id == 2 && no_of_coke > 0)
    {
        if (inserted > COKE_COST)
        {
            inserted -= COKE_COST;
            no_of_coke -= 1;
        }
    }
}

void get_change()
{
    if (check_have_change())
    {
        while (inserted > 10 && ten_pennies > 0)
        {
            inserted -= 10;
            ten_pennies -=1;
        }

        while (inserted > 5 && five_pennies > 0)
        {
            inserted -= 5;
            five_pennies -=1;
        }
        while (inserted > 2 && two_pennies > 0)
        {
            inserted -= 2;
            two_pennies -=1;
        }

        while (inserted > 1 && one_pennies > 0)
        {
            inserted -= 1;
            one_pennies -=1;
        }
    }
}
```

Bibliography

[Ant89] J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In *Proceedings of the 3rd International Conference on Genetic Algorithms and Their Applications*, pages 86–91, San Mateo, California, USA, 1989. Morgan Kaufmann.

[Bac96] T. Back. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, 1996.

[Bar02] A. Baresel. Private communication, 2002.

[BBHK04] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 43–52, Boston, Massachusetts, USA, 2004. ACM.

[Bei90] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 2nd edition, 1990.

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT - A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–244. ACM Press, 1975.

[BHS91] T. Back, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In L. Booker and R. Belew, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9, San Diego, California, USA, 1991. Morgan Kaufmann.

[Bot02] L. Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337 – 1342, New York, USA, 2002. Morgan Kaufmann.

[BPS03] A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2428 – 2441, Chicago, USA, 2003. Springer-Verlag.

[BS03] A. Baresel and H. Sthamer. Evolutionary testing of flag conditions. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2442 – 2454, Chicago, USA, 2003. Springer-Verlag.

[BSS02] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1329–1336, New York, USA, 2002. Morgan Kaufmann.

[BW03] O. Buehler and J. Wegener. Evolutionary functional testing of an automated parking system. In *International Conference on Computer, Communication and Control Technologies and The 9th International Conference on Information Systems Analysis and Synthesis*, Orlando, Florida, USA, 2003.

[CC99] C. A. Coello Coello. An updated survey of evolutionary multiobjective optimization techniques: State of the art and future trends. In *1999 Congress on Evolutionary Computation*, pages 3–13. IEEE Service Center, 1999.

[CDG99] D. Corne, M. Dorigo, and F. Glover, editors. *New Ideas in Optimization.* McGraw-Hill, 1999.

[CDH$^+$00] J. C. Corbett, M. Dwyer, J. Hatcliff, Robby Laubach S. Pasareanu, C., and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–449, 2000.

[Cla76] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[cyg04] Cygwin environment, http://www.cygwin.com, 2004.

[Dav96] L. Davis. *Handbook of Genetic Algorithms.* International Thomson Computer Press, 1996.

[Deb00] K. Deb. Multi-objective evolutionary optimization: Past, present and future. In *Proceedings of the Fourth International Conference on Adaptive Computing in Design and Manufacture (ACDM'2000)*, pages 225–236, University of Plymouth, UK, 2000. Springer, London.

[DG91] K. Deb and D. Goldberg. A comparative analysis of selection schemes used in genetic algorithms. In G. J. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, San Mateo, California, USA, 1991.

[DHJ$^+$01] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, Toronto, Canada, 2001.

[DIS99] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. 29(7):577–603, 1999.

[DO91] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–909, 1991.

[FK96a] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[FK96b] R. Ferguson and B. Korel. Generating test data for distributed software using the chaining approach. *Information and Software Technology*, 38(5):343–353, 1996.

[FOW87] J. Ferrante, K. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[gea04] GEATbx - Genetic and Evolutionary Algorithm Toolbox, http://www.geatbx.com, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, New York, 1979.

[GN97] M. J. Gallagher and V. L. Narasimhan. ADTEST: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473 – 484, 1997.

[Gol89] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

[Gor91] S. M. Gorges. Explicit parallelism of genetic algorithms through population structures. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 150–159. Springer-Verlag, 1991.

[GPY02] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science vol. 2280*, pages 357 – 370. Springer-Verlag, 2002.

[Gro01] H.-G. Gross. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information and Software Technology*, 43(14):855–862, 2001.

[Har02] M. Harman. Deceptive objective function example, presented at the Search-Based Software Engineering Workshop, September 2002.

[HFH$^+$02] M. Harman, C. Fox, R. Hierons, L. Hu, S. Danicic, and J. Wegener. VADA: A transformation-based system for variable dependence analysis. In *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Montreal, Canada, 2002.

[HHH$^+$02] M. Harman, L. Hu, R. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1359–1366, New York, USA, 2002. Morgan Kaufmann.

[HHH$^+$04] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[HHZ$^+$02] M. Harman, L. Hu, X. Zhang, M. Munro, J. J. Dolado, M. C. Otero, and J. Wegener. A post-placement side-effect removal algorithm. In *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, pages 2–11, Montreal, Canada, 2002.

[HHZM01] M. Harman, L. Hu, X. Zhang, and M. Munro. Side-effect removal transformation. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension (IWPC 2001)*, pages 310–319, Toronto, Canada, 2001. IEEE Computer Society Press.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, 1975.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[HS01] G. Holzmann and M. Smith. Software model checking: Extracting verification models from source code. *Software Testing, Verification and Reliability*, 11(2):65–79, 2001.

[JSE96] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[JSYE95] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management*, pages 435–444, Seville, Spain, 1995.

[KAY96] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pages 71–80, 1996.

[KGV83] S. Kirkpatrick, C. D. Gellat, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Kin75] J. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, pages 228 – 233. ACM Press, 1975.

[Kin76] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[KLV$^+$96] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. Object state testing and fault analysis for reliable software systems. In *Proceedings of the 7th International Symposium on Software Reliability Engineering*, pages 76–85, White Plains, New York, USA, 1996.

[Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[Kor92] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[Kor96] B. Korel. Automated test generation for programs with procedures. In *International Symposium on Software Testing and Analysis (ISSTA 1996)*, pages 209–215, San Diego, California, USA, 1996.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[KSGH94] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proceedings of the 18th Annual International Computer Software and Applications Conference*, pages 222–227, Taipei, Taiwan, 1994.

[MH03] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003), Lecture Notes in Computer Science vol. 2724*, pages 2488–2497, Chicago, USA, 2003. Springer-Verlag.

[Mit96] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1996.

[MMS01] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[MRR$^+$53] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[MS76] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[MSV93] H. Muhlenbein and D Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993.

[Mye79] G. J. Myers. *The Art of Software Testing.* Wiley, 1979.

[OJP99] A. J. Offutt, Z. Jin, and J. Pan. The dynamic domain reduction procedure for test data generation. *Software - Practice and Experience*, 29(2):167–193, 1999.

[OVW98] M. O'Sullivan, S. Vössner, and J. Wegener. Testing temporal correctness of real-time systems - a new approach using genetic algorithms and cluster analysis. In *Proceedings of the 6th European Conference on Software Testing, Analysis and Review (EuroSTAR 1998)*, Munich, Germany, 1998.

[PD00] L. L. Peterson and B. S. Davie. *Computer Networks - A Systems Approach.* Morgan Kaufmann, 2000.

[PHP99] R. Pargas, M. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[PN98] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 134–143, Madrid, Spain, 1998. IEEE Computer Society Press.

[Ree95] C. R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* McGraw-Hill, 1995.

[RHC76] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[Rop97] M. Roper. Computer aided software testing using genetic algorithms. In *10th International Software Quality Week*, San Francisco, USA, 1997.

[SP94] M. Srinivas and Lalit M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, 1994.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[SVM94] D. Schlierkamp-Voosen and H. Mühlenbein. Strategy adaptation by competing subpopulations. In *Parallel Problem Solving from Nature*, pages 199–208. Springer-Verlag, 1994.

[SVM96] D. Schlierkamp-Voosen and H. Mühlenbein. Adaptation of population sizes by competing subpopulations. In *Proceedings of the IEEE Conference on Evolutionary Computation*, pages 330–335, Piscataway, New Jersey, USA, 1996. IEEE Press.

[SWM91] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H. P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 176–185, Berlin, 1991. Springer-Verlag.

[Tan89] R. Tanese. Distributed genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 434–439, San Mateo, California, USA, 1989. Morgan Kaufmann.

[TCM98a] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Software Engineering Notes, Issue 23, No. 2, Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 73–81, 1998.

[TCM98b] N. Tracey, J. Clark, and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications*, pages 169–180. Dept of Computer Science, University of Witwatersrand, Johannesburg, South Africa, 1998.

[TCMM98] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA, 1998. IEEE Computer Society Press.

[TCMM00] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test data generation for exception conditions. *Software - Practice and Experience*, 30(1):61–79, 2000.

[Ton04] P. Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128, Boston, USA, 2004. ACM Press.

[Tra00] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software.* PhD thesis, University of York, 2000.

[Wat95] A. Watkins. The automatic generation of test data using genetic algorithms. In *Proceedings of the Fourth Software Quality Conference*, pages 300–309, 1995.

[WBP02] J. Wegener, K. Buhr, and H. Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1233–1240, New York, USA, 2002. Morgan Kaufmann.

[WBS01] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[Weg03] J. Wegener. Private communication, 2003.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[WG98] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, 1998.

[WGG$^+$96] J. Wegener, K. Grimm, M. Grochtmann, H. Sthamer, and B. Jones. Systematic testing of real-time systems. In *Proceedings of the 4th European Conference on Software Testing, Analysis and Review (EuroSTAR 1996)*, Amsterdam, Netherlands, 1996.

[Whi89] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the 3rd International Conference on*

Genetic Algorithms, pages 116–121, San Mateo, California, USA, 1989. Morgan Kaufmann.

[Whi94] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[Whi99] D. Whitley. A free lunch proof for gray versus binary encodings. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 726–733, Orlando, Florida, USA, 1999. Morgan Kaufmann.

[Whi01] D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, 2001.

[WPS99] J. Wegener, H. Pohlheim, and H. Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR 1999)*, Barcelona, Spain, 1999.

[WPS00] J. Wegener, R. Pitschinetz, and H. Sthamer. Automated testing of real-time tasks. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, 2000.

[WRDM96] D. Whitley, S. B. Rana, J. Dzubera, and K. E. Mathias. Evaluating evolutionary algorithms. *Artificial Intelligence*, 85(1-2):245–276, 1996.

[WSJE97] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997.

[XES+92] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, Toulouse, France, 1992.