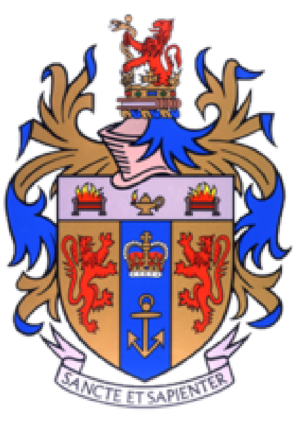
Search–Based Testing



Kiran Lakhotia

Department of Computer Science

King's College London

Submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy at King's College London

October 2009

Abstract

This thesis is concerned with the problem of automatic test data generation for structural testing criteria, in particular the branch coverage adequacy criterion, using search–based techniques. The primary objective of this thesis is to advance the current state–of–the–art in automated search–based structural testing. Despite the large body of work within the field of search–based testing, the accompanying literature remains without convincing solutions for several important problems, including: support for pointers, dynamic data structures, and loop–assigned flag variables. Furthermore, relatively little work has been done to extend search–based testing to multi–objective problem formulations.

One of the obstacles for the wider uptake of search–based testing has been the lack of publicly available tools, which may have contributed to the lack of empirical studies carried out on real–world systems. This thesis presents AUSTIN, a prototype structural test data generation tool for the C language. The tool is built on top of the CIL framework and combines a hill climber with a custom constraint solver for pointer type inputs. AUSTIN has been applied to five large open source applications, as well as eight non–trivial, machine generated C functions drawn from three real–world embedded software modules from the automotive sector. Furthermore, AUSTIN has been compared to a state–of–the–art Evolutionary Testing Framework and a dynamic symbolic execution tool, CUTE. In all cases AUSTIN was shown to be competitive, both in terms of branch coverage and efficiency.

To address the problem of loop–assigned flags, this thesis presents a testability transformation along with a tool that transforms programs with loop–assigned flags into flag–free equivalents, so that existing search–based test data generation approaches can successfully be applied.

The thesis concludes by introducing multi–objective branch coverage. It presents results from a case study of the twin objectives of branch coverage and dynamic memory consumption for both real and synthetic programs.

Acknowledgements

I would like to thank my supervisor Professor Mark Harman for providing this opportunity, and for his endless support and advice throughout this PhD. I would also like to thank my second supervisor, Dr Kathleen Steinhöfel, for her support and comments, especially during the early stages of this PhD.

In addition, I am very grateful to Berner & Mattner GmbH, and in particular Dr Joachim Wegener, for providing me with the opportunity to evaluate AUSTIN on real–world code. Thanks also to Laurence Tratt for his helpful comments on the thesis and Arthur Baars for taking the time to explain the Evolutionary Testing Framework. I would further like to thank all the co–authors of the papers I have published, partly as a result of this work, for all their hard work and all the anonymous referees for their comments and feedback, as these have been extremely beneficial. Thanks also to Dr Phil McMinn for making the IGUANA testing framework available to me and to all my colleagues in CREST for their support and discussions.

The achievements of this thesis would not have been possible without the support of my parents who provided me with the opportunity to pursue my interest in computer science, and my partner, Helen Hudson, who I would like to thank for her love, support, and patience during the three years of this project.

Declaration

I herewith declare that I have produced this thesis without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

The thesis work was conducted from September 2006 to September 2009 under the supervision of Professor Mark Harman at King's College London. Some of this work has been accepted to appear in the refereed literature:

1. D. Binkley, M. Harman, K. Lakhotia, FlagRemover: A Testability Transformation For Loop Assigned Flags, *Transactions on Software Engineering and Methodology (TOSEM)*, To appear.
2. K. Lakhotia, P. McMinn, M. Harman, Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet?, *4th Testing Academia and Industry Conference - Practice and Research Techniques (TAIC PART 09)*, Windsor, UK, 4th–6th September 2009, To appear.
3. M. Harman, K. Lakhotia, P. McMinn, A Multi–Objective Approach to Search-Based Test Data Generation, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, London, UK, July 7–11, 2007, pp. 1098-1105, ACM Press.

In addition to the above, during the programme of study, I have published the following papers, the work of which does not feature in this thesis:

1. K. Lakhotia, M. Harman, P. McMinn, Handling Dynamic Data Structures in Search Based Testing, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*, Atlanta, USA, July 12–16, 2008, pp. 1759–1766, ACM Press.[1]

[1] This work was a precursor to the work done in Chapter 3.

2. M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, J. Wegener, The Impact of Input Domain Reduction on Search-Based Test Data Generation, *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2007)*, Cavtat, Croatia, September 3–7, 2007, pp. 155–164, ACM Press.

Contents

List of Figures ix

List of Tables xvii

1 Introduction 1

1.1 The Problem Of The Thesis: Practical Challenges For Automated Test Data Generation . . . 3

1.2 Aims and Objectives . . . 4

1.3 Contributions Of The Thesis . . . 5

1.4 Overview Of The Structure Of The Thesis . . . 6

1.5 Definitions . . . 7

1.5.1 Input Domain . . . 7

1.5.2 Control and Data Dependence . . . 8

2 Literature Review 13

2.1 Search–Based Testing . . . 13

2.1.1 Genetic Algorithms . . . 17

2.1.2 Hill Climb . . . 20

2.1.3 Fitness Functions . . . 21

2.2 Static Analysis Based Testing . . . 24

2.2.1 Symbolic Execution . . . 25

2.2.2 Dynamic Symbolic Execution . . . 27

3 Augmented Search–Based Testing 35

3.1 Introduction . . . 35

3.2 AUSTIN . . . 37

3.2.1 Floating Point Variables 37
3.2.2 Example 40
3.2.3 Code Preparation 41
3.2.4 Symbolic Rewriting 44
3.2.5 Pointer Inputs 49
3.2.5.1 Limitations of Pointer Rules 54
3.2.6 The Algorithm 58
3.2.7 Input Initialization 65
3.2.7.1 Limitations of Input Types 66
3.2.8 Usage 69
3.3 Evolutionary Testing Framework 70
3.4 Empirical Study 71
3.4.1 Experimental Setup 75
3.4.2 Evaluation 78
3.4.3 Threats to Validity 84
3.5 Conclusion 86

4 An Empirical Investigation Comparing AUSTIN and CUTE 87

4.1 Introduction 87
4.2 CUTE 89
4.3 Motivation and Research Questions 89
4.4 Empirical Study 91
4.4.1 Test Subjects 91
4.4.2 Experimental Setup 92
4.4.3 Answers to Research Questions 93
4.4.3.1 CUTE–specific issues 101
4.4.3.2 AUSTIN–specific issues 103
4.5 Threats to Validity 105
4.6 Discussion: Open Problems in Automated Test Data Generation 106
4.7 Related Work 107
4.8 Conclusion 108

5 Testability Transformation **109**
5.1 Introduction . . . 109
5.2 Background . . . 111
5.2.1 The Flag Problem . . . 111
5.2.2 Testability Transformation . . . 112
5.3 The Flag Replacement Algorithm . . . 112
5.4 Implementation . . . 118
5.4.1 Definition of Loop–Assigned Flag . . . 118
5.4.2 Flag Removal . . . 120
5.4.3 Runtime . . . 121
5.4.4 Limitations . . . 121
5.5 Empirical Algorithm Evaluation . . . 123
5.5.1 Synthetic Benchmarks . . . 123
5.5.2 Open Source and Daimler Programs . . . 128
5.6 Relevant Literature For the Loop–Assigned Flag Problem . . . 137
5.7 Conclusion . . . 139

6 Multi–Objective Test Data Generation **141**
6.1 Introduction . . . 141
6.2 Background . . . 143
6.3 Implementation . . . 145
6.3.1 Pareto Genetic Algorithm Implementation . . . 146
6.3.2 Weighted Genetic Algorithm Implementation . . . 147
6.4 Experimental Setup . . . 148
6.5 Case Study Results . . . 148
6.6 Discussion . . . 151
6.7 Conclusion . . . 156

7 Conclusions and Future Work **157**
7.1 Summary of Achievements . . . 157
7.2 Summary of Future Work . . . 160

A Abbreviations and Acronyms **165**

References **167**

List of Figures

1.1 Code template in the left column is used to illustrate search–based test data generation. In the right column is the CFG of the code in the left column. The CFG is annotated with the *approach level* for node 4. . . . 9

1.2 A code example with the corresponding CFG on the right illustrating how data flow information is not captured explicitly in a CFG. The true branch of node 4 is data dependent on node 3, but not control dependent on any other node except the start node. 11

2.1 Example used to demonstrate Korel's goal oriented (Kor92) test data generation approach. 16

2.2 A typical evolutionary algorithm for testing. 18

2.3 Example illustrating how the crossover operator in a GA combines parts of two candidate solutions, *Candidate String 1* and *Candidate String 2*, to form an offspring representing the desired string. 19

2.4 Example used to illustrate how machine semantics differ from mathematical principles over reals and rational numbers. Arithmetic suggests that the path $\langle 1, 2, 3, 4, 5 \rangle$ is infeasible, when in practice there exist values for x that satisfy this path. 27

2.5 Example for demonstrating dynamic symbolic execution. The branch predicate is non–linear. 28

2.6 Example for demonstrating pointer handling in CUTE. 30

2.7 Code example used to illustrate how the EVACON framework (IX07) is able to optimize both, the method call sequences required to test the function m, as well as the method parameters to achieve coverage of m. 33

LIST OF FIGURES

3.1 Example C code used for demonstrating how AUSTIN combines custom constraint solving rules for pointer inputs with an AVM for other inputs to a function under test. The goal is to find an input which satisfies the condition at node 2. 38

3.2 Pseudo code illustrating the method for checking if the search is stuck at a local optimum. Part of this method also includes the steps for varying the accuracy of floating point variables as described in Section 3.2.1. . . 39

3.3 Example for demonstrating how AUSTIN optimizes floating point type inputs. Previous work (MH06; LMH09) commonly fixed the accuracy for floating point variables in the AVM to 1 or 2 decimal places. Such a strategy suffices to generate inputs to satisfy the branch predicate in the left column, but is unlikely to succeed for the branch predicate in the right column. AUSTIN is able to automatically satisfy the branch predicates in both the left and right columns. 40

3.4 Two examples illustrating how CIL transforms compound predicates into single predicate equivalents. The transformation works for both, predicates in conditional statements (see (a)) and in–line predicates (see (b)). 43

3.5 Ocaml type declarations in AUSTIN used to denote symbolic lvalues and symbolic terms. A symbolic lvalue is used to represent an input variable to the function under test, while symbolic terms provide a wrapper for CIL expressions. Symbolic terms only consist of operations involving constants or input variables to the function under test. 45

3.6 Code snippet illustrating the Ocaml class for rewriting CIL expressions in terms of input variables. The class extends CIL's default visitor, which traverses a CIL tree without modifying anything. The method `vexpr` is invoked on each occuring (CIL) expression. The subtrees are the sub–expressions, the types (for a cast or `sizeof` expression) or the variable use. 47

3.7 The example in the top half demonstrates how symbols are updated in a symbolic store using the visitor class from Figure 3.6. The bottom row shows how AUSTIN approximates symbolic expressions with the help of concrete runtime values instead of performing symbolic pointer arithmetic. The expression `((struct item **)(tmp1 + 8))` is replaced by the expression denoting the address of `one->next`, *i.e.* `&one->next`, instead of rewriting the expression as `((struct item **)(((unsigned long)one) + 8))` using the visitor class from Figure 3.6. 48
3.8 Code snippet used as a running example to explain AUSTIN's pointer solving rules. 51
3.9 An example illustrating how AUSTIN builds up an equivalence graph of symbolic terms over successive executions. The graph is used to instantiate the concrete pointer type inputs to the function under test. The example shows the evolution of the graph when node 5 has been selected as target. 53
3.10 Pseudo code illustrating the algorithm for constructing an equivalence graph. The graph forms the basis for deriving values assigned to pointer inputs of a function under test. 55
3.11 Pseudo code illustrating the algorithm for assigning concrete values to pointers. The manner in which new or existing memory locations are assigned to a function's pointer inputs depends on the structure of the equivalence graph. 56
3.12 Pseudo code showing the top level search algorithm implemented in AUSTIN. It extends a hill climber to include constraint solving rules for pointer inputs to a function under test. The steps for handling numerical type inputs to a function under test are based on the alternating variable method introduced by Korel (Kor90). 59
3.13 Pseudo code illustrating the algorithm for performing exploratory moves which are part of the alternating variable method. 61
3.14 Pseudo code illustrating the algorithm for performing pattern moves in the alternating variable method. Pattern moves are designed to speed up the search by accelerating towards an optimum. 62

3.15 Pseudo code illustrating the algorithm for optimising n–valued enumeration type variables, variables that can only take on n discrete values. Whenever an enumeration type variable is to be optimized, the algorithm iterates over all n values (in the order they have been declared in the source code). 63

3.16 Pseudo code illustrating the method for checking if AUSTIN should use its custom constraint solving rules for pointers, or the alternating variable method to satisfy the constraint in a branching node. 64

3.17 Pseudo code illustrating the reset operation performed by AUSTIN at every random restart. 65

3.18 Pseudo code illustrating the entry point for initializing a candidate solution (*i.e.* the inputs to the function under test) with concrete values. . 66

3.19 Pseudo code illustrating the method assigning concrete values to an input variable of the function under test. 67

3.20 This group of methods shows the pseudo code for assigning values to inputs of the function under test, based on their type declaration. 68

3.21 Pseudo code illustrating how AUSTIN instantiates its map of concrete values m_C and address map m_A at the start of every execution of the function under test. The maps are also updated during the dynamic execution of the function under test. 69

3.22 Example showing how the ETF initializes pointers to primitive types. The chromosome only shows the parts which relate to the two pointer inputs `p` and `q`. The first component of each pair denotes an index (of a variable in a pool) while the second component denotes a value. 72

3.23 Average branch coverage of the ETF versus AUSTIN ($N \geq 30$). The $y - axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$). 79

3.24 Average branch coverage of random search versus AUSTIN ($N \geq 29$). The $y - axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$). 81

3.25 Average number of fitness evaluations (normalized) for ETF versus AUSTIN ($N \geq 30$). The $y - axis$ shows the normalized average number of fitness evaluations for each tool relative to the ETF (shown as 100%) for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$). 82

3.26 Average number of fitness evaluations (normalized) for random versus AUSTIN ($N \geq 29$). The $y - axis$ shows the normalized average number of fitness evaluations for each tool relative to the random search (shown as 100%) for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$). 84

4.1 Branch coverage for the test subjects with CUTE and AUSTIN. CUTE explicitly explores functions called by the function under test, whereas AUSTIN does not. Therefore the graph 4.1(a) counts only branches covered in each function tested individually. Graph 4.1(b) counts branches covered in the function under test and branches covered in any functions called. Graph 4.1(c) is graph 4.1(b) but with certain functions that CUTE cannot handle excluded. 94

4.2 An example used to illustrate how a random restart in AUSTIN can affect the runtime of a function under test, and thus the test data generation process. Very large values of `upb` may have a significant impact on the wall clock time of the test data generation process. 98

5.1 This figure uses three fitness landscapes to illustrate the effect flag variables have on a fitness landscape, and the resulting 'needle in a haystack' problem. The figure has been taken from the paper by Binkley *et al.* (BHLar). 110

5.2 An example program before and after applying the coarse and fine–grain transformations. The figures also shows part of the function for computing local fitness. 114

5.3 The transformation algorithm. Suppose that `flag` is assigned `true` outside the loop and that this is to be maintained. 116

5.4 Results over ten runs of the evolutionary search for each of the two transformation approaches. The graphs have been taken from the paper by Baresel *et al.* (BBHK04). 125

5.5 Results over ten runs of the evolutionary search for the fine–grained transformation approach close–up. The graph has been taken from the paper by Baresel *et al.* (BBHK04). 126

5.6 Results over ten runs of the alternating variable method for the 'no transformation' and fine–grained transformation approaches. 127

5.7 Chart of data from second empirical study. 136

6.1 Results of the branch coverage and memory allocation achieved by three different algorithms: a random search, a Pareto GA and, weighted GA. 152

6.2 Final Pareto fronts produced for targets 1T and 1F. The upper point on the y–axis represents the 'ideal' solution for target 1F. As can be seen, once the branch has been reached, a single solution will dominate all others because it is the only branch allocating memory. When attempting to cover target 1T on the other hand, the Pareto optimal set potentially consists of an infinite number of solutions. The graph combines five runs which reveal little variance between the frontlines produced. Interestingly, the 'ideal' point for target 1F corresponds to the maximum value contained within the Pareto optimal set for target 1T with respect to memory allocation. 154

6.3 The table at the bottom presents the Pareto optimal sets for each 'sub–goal' of the example function above used in Case Study 2. It combines the results collected over five runs and illustrates that it is often not possible to generate a Pareto frontline when considering branch coverage and memory allocation as a MOP. Although the amount of dynamic memory allocated depends on the input parameters, it is constant for all branches. As a result one solution will dominate all others with respect to a particular target. 155

7.1 Code example to illustrate ideas for future work. 162

List of Tables

2.1 Branch functions for relational predicates introduced by Korel (Kor90). 14

2.2 Branch distance measure introduced by Tracey (Tra00) for relational predicates. The value K is a failure constant which is always added if a term is false. 23

2.3 Objective functions introduced by Tracey (Tra00) for conditionals containing logical connectives. z is one of the objective functions from Table 2.2. 23

2.4 Dynamic data structure creation according to individual constraints encountered along the path condition for CUTE. 31

3.1 Case studies. LOC refers to the total preprocessed lines of C source code contained within the case studies. The LOC have been calculated using the CCCC tool (Lit01) in its default setting. Table 3.2 shows the individual per–function LOC metric. 72

3.2 Test subjects. The LOC have been calculated using the CCCC tool (Lit01) in its default setting. The number of input variables counts the number of independent input variables to the function, *i.e.* the member variables of data structures are all counted individually. 74

3.3 Summary of functions with a statistically significant difference in the branch coverage achieved by AUSTIN and the ETF. The columns Std-Dev indicate the standard deviation from the mean for ETF and AUSTIN. The t value column shows the degrees of freedom (value in brackets) and the result of the t–test. A p value of less than 0.05 means there is a statistically significant difference in the mean coverage between ETF and AUSTIN. 78

3.4 Summary of functions with statistically significant differences in the number of fitness evaluations used. The columns StdDev indicate the standard deviation from the mean for ETF and AUSTIN. The t value column shows the degrees of freedom (value in brackets) and the result of the t–test. A p value of less than 0.05 means there is a statistically significant difference in the mean number of fitness evaluations between ETF and AUSTIN. 83

4.1 Details of the test subjects used in the empirical study. In the 'Functions' column, 'Non–trivial' refers to functions that contain branching statements. 'Top–level' is the number of non–trivial, public functions that a test driver was written for, whilst 'tested' is the number of functions that were testable by the tools (*i.e.* top–level functions and those that could be reached interprocedurally). In the 'Branches' column, 'tested' is the number of branches contained within the tested functions. 90

4.2 Test subjects used for comparing wall clock time and interprocedural branch coverage. 95

4.3 Interprocedural branch coverage for the sample branches from Table 4.2 and the wall clock time taken to achieve the respective levels of coverage. 96

4.4 Errors encountered for test data generation for the sample of branches listed in Table 4.3. 99

5.1 Runtime of the transformation (in seconds) for the test subjects as reported by the `time` utility. The measurements are averaged over five runs. The column *real* refers to the wall clock time, *user* refers to the time used by the tool itself and any library subroutines called, while *sys* indicates the time used by system calls invoked by the tool. 122

5.2 Test subjects for the evaluation of the transformation algorithm. 129

5.3 Branch coverage results from empirical study of functions extracted from open source software. 134

5.4 Fitness function evaluations from empirical study of functions extracted from open source software. 135

6.1 The table shows the branches covered by the weighted GA and not the Pareto GA, or vice versa. The 'distance' measure illustrates how close the best solution came to traversing the target branch. It combines the normalized branch distance and the approach level. A distance of 0 indicates a branch has been covered. These results were obtained during Case Study 1. 156

Chapter 1

Introduction

In recent years there has been a particular rise in the growth of work on Search–Based Software Testing (SBST) and, specifically, on techniques for generating test data that meet structural coverage criteria such as branch– or modified condition / decision coverage. The field of search–based software engineering reformulates software engineering problems as optimization problems and uses meta–heuristic algorithms to solve them. Meta–heuristic algorithms combine various heuristic methods in order to find solutions to computationally hard problems where no problem specific heuristic exists. SBST was the first software engineering problem to be attacked using optimization (MS76) and it remains the most active area of research in the search–based software engineering community.

Software testing can be viewed as a sequence of three fundamental steps:

1. The design of test cases that are effective at revealing faults, or which are at least adequate according to some test adequacy criterion.

2. The execution of these test cases.

3. The determination of whether the output produced is correct.

Sadly, in current testing practice, often the only fully automated aspect of this activity is test case execution. The problem of determining whether the output produced by the program under test is correct cannot be automated without an oracle, which is seldom available. Fortunately, the problem of generating test data to achieve widely

used notions of test adequacy *is* an inherently automatable activity, especially when considering structural coverage criteria as is the case in this thesis.

Such automation promises to have a significant impact on testing, because test data generation is such a time–consuming and laborious task. A number of hurdles still remain before automatic test data generation can be fully realized, especially for the C programming language. This thesis aims to overcome some of these and thus concerns itself with advancing the current state–of–the–art in automated search–based structural testing. It only focuses on the branch coverage test adequacy criterion, a criterion required by testing standards for many types of safety critical applications (Bri98; Rad92).

Since the early 1970's two widely studied schools of thought regarding how to best automate the test data generation process have developed: dynamic symbolic execution and search–based testing.

Dynamic symbolic execution (SMA05; GKS05; CE05) originates in the work of Godefroid *et al.* on Directed Random Testing (GKS05). It formulates the test data generation problem as one of finding a solution to a constraint satisfaction problem, the constraints of which are produced by a combination of dynamic and symbolic (Kin76) execution of the program under test. Concrete execution drives the symbolic exploration of a program, and dynamic variable values obtained by real program execution can be used to simplify path constraints produced by symbolic execution.

Search–based testing (McM04) formulates the test data adequacy criteria as objective functions, which can be optimized using Search–Based Software Engineering (CDH+03; Har07). The search–space is the space of possible inputs to the program under test. The objective function captures the particular test adequacy criterion of interest. The approach has been applied to several types of testing, including functional (WB04) and non–functional (WM01) testing, mutation testing (BFJT05), regression testing (YH07), test case prioritization (WSKR06) and interaction testing (CGMC03). However, the most studied form of search–based testing has been structural test data generation (MS76; Kor90; RMB+95; PHP99; MMS01; WBS01; Ton04; HM09). Despite the large body of work on structural search–based testing, the techniques proposed to date do not fully extend to pointers and dynamic data structures. Perhaps one reason is the lack of publicly available tools that provide researchers with facilities to perform search–based structural testing. This thesis introduces such a tool, AUSTIN. It uses a variant of

Korel's (Kor90) alternating variable method and augments it with techniques adapted from dynamic symbolic execution (GKS05; CE05; SMA05; TdH08) in order to handle pointers and dynamic data structures.

The next section will discuss in detail the problems this thesis addresses. An overview of the relevant literature for this thesis is presented in Chapter 2.

1.1 The Problem Of The Thesis: Practical Challenges For Automated Test Data Generation

Previous work on search–based test data generation has generally considered the input to the program under test to be a fixed–length vector of input values, making it a well–defined and fixed–size search–space, or has been based on data–flow–analysis and backtracking (Kor90). However, such analysis is non–trivial in the presence of pointers [1] and points–to analysis is computationally expensive.

The approach presented in this thesis incorporates elements from symbolic execution to overcome this problem. Symbolic execution is a static source code analysis technique in which program paths are described as a constraint set involving only the input parameters of a program (Kin76). The key idea behind the proposed approach is to model all inputs to a program, including memory locations, as scalar symbolic variables, and to perform a symbolic execution of a single path in parallel to a concrete execution as part of the search–based testing process. The approach uses a custom constraint solver designed for constraints over pointer inputs, which can also be used to incrementally build dynamic data structures, resulting in a variable–size search–space.

Besides pointer and dynamic data structures, boolean variables (*i.e.* flag variables) remain a problem for many search–based testing techniques. In particular loop–assigned flags, a special class of flag variables whose definition occurs within the body of a loop and whose use is outside that loop, have not been addressed by the majority of work investigating the flag problem. This thesis therefore also considers a testability transformation for loop–assigned flags and its effect on search–based testing.

Finally, there has been little work on multi–objective branch coverage. In many scenarios a single–objective formulation is unrealistic; testers will want to find test sets

[1] Pointers are variables which can hold the address of another program variable. In particular they can be used to manipulate specific memory locations either in a program's stack or heap.

that meet several objectives simultaneously in order to maximize the value obtained from the inherently expensive process of running the test cases and examining the output they produce. For example, the tester may wish to find test cases that are more likely to be fault–revealing, or test cases that combine different non–subsuming coverage based criteria. The tester might also be concerned with test cases that exercise the usage of the stack or the heap, potentially revealing problems with the stack size or with memory leaks and heap allocation problems. There may also be additional domain–specific goals the tester would like to achieve, for instance, exercising the tables of a database in a certain way, or causing certain implementation states to be reached.

In any such scenario in which the tester has additional goals over and above branch–coverage, existing approaches represent an over simplification of the problem in hand. A multi–objective optimization approach would be a more realistic approach. This thesis takes a first step by considering the formulation of a multi–objective branch–coverage test adequacy criterion.

1.2 Aims and Objectives

The aims of this thesis are the following:

1. Advance the capabilities of the current state–of–the–art search–based testing techniques, extending them so they can handle pointers and dynamic data structures.

2. Perform a thorough empirical investigation evaluating the extended search–based strategy against a concolic testing approach. The study will aim to provide a concrete domain of programs for which the approach will either be adequate or inadequate, and any insight gained will be generalisable to instances of programs in that domain.

3. Empirically investigate the use of a testability transformation for the loop–assigned flag problem in search–based–testing.

4. Investigate the use of search–based testing in multi–objective test data generation problems.

1.3 Contributions Of The Thesis

The contributions of this thesis are:

1. AUSTIN, a fully featured search–based software testing tool for C.
2. An empirical study that evaluates AUSTIN on automotive systems for branch coverage. The results support the claim that AUSTIN is both efficient and effective when applied to machine generated code.
3. An empirical study which determines the level of code coverage that can be obtained using CUTE and AUSTIN on the complete source code of five open source programs. Perhaps surprisingly, the results show that only modest levels of coverage are possible at best, and there is still much work to be done to improve test data generators.
4. An assessment, based on the empirical study, of where CUTE and AUSTIN succeeded and failed, and a discussion and detailed analysis of some of the challenges that remain for improving automated test data generators to achieve higher levels of code coverage.
5. A testability transformation algorithm which can handle flags assigned in loops.
6. Two empirical studies evaluating the transformation algorithm. They show that the approach reduces test effort and increases test effectiveness. The results also indicate that the approach scales well as the size of the search–space increases.
7. A first formulation of test data generation as a multi–objective problem. It describes the particular goal oriented nature of the coverage criterion, showing how it presents interesting algorithmic design challenges when combined with the non goal oriented memory consumption criterion.
8. A case study, the result of which confirm that multi–objective search algorithms can be used to address the problem, by applying the ‘sanity check’ that search–based approaches outperform a simple multi–objective random search.

1.4 Overview Of The Structure Of The Thesis

Chapter 2 surveys the literature in the field of search–based structural testing. The chapter describes the two most commonly used algorithms in search–based testing, genetic algorithms and hill climbing, before moving on to describe the fitness functions used in search–based testing for the branch coverage adequacy criterion. Next, the chapter addresses testing from a static analysis point of view. Symbolic execution was one of the earliest methods used in testing. More recently researchers have investigated ways to combine symbolic execution with dynamic analyses in a field known as dynamic symbolic execution.

Chapter 3 presents AUSTIN, a prototype structural test data generation tool for the C language. The tool is built on top of the CIL framework and combines a hill climber with a custom constraint solver for pointer type inputs. An empirical study is presented in which AUSTIN's effectiveness and efficiency in generating test data is compared with the state–of–the–art Evolutionary Testing Framework (ETF) structural test component, developed within the scope of the EU–funded EvoTest project. The study also includes a comparison with the ETF configured to perform a random search. The test objects consisted of eight non–trivial C functions drawn from three real–world embedded software modules from the automotive sector and implemented using two popular code–generation tools. For the majority of the functions, AUSTIN is at least as effective (in terms of achieved branch coverage) as the ETF and is considerably more efficient.

Chapter 4 presents an empirical study applying a concolic testing tool, CUTE, and AUSTIN to the source code of five large open source applications. Each tool is applied 'out of the box'; that is without writing additional code for special handling of any of the individual subjects, or by tuning the tools' parameters. Perhaps surprisingly, the results show that both tools can only obtain at best a modest level of code coverage. Several challenges remain for improving automated test data generators in order to achieve higher levels of code coverage, and these are summarized within the chapter.

Chapter 5 introduces a testability transformation along with a tool that transforms programs with loop–assigned flags into flag–free equivalents, so that existing search–based test data generation approaches can successfully be applied. The chapter presents the results of an empirical study that demonstrates the effectiveness and efficiency of the testability transformation on programs including those made up of open source and industrial production code, as well as test data generation problems specifically created to denote hard optimization problems.

Chapter 6 introduces multi–objective branch coverage. The chapter presents results from a case study of the twin objectives of branch coverage and dynamic memory consumption for both real and synthetic programs. Several multi–objective evolutionary algorithms are applied. The results show that multi–objective evolutionary algorithms are suitable for this problem. The chapter also illustrates how a Pareto optimal search can yield insights into the trade–offs between the two simultaneous objectives.

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

1.5 Definitions

This section contains common definitions used throughout the thesis. They have been added to make the thesis self contained.

1.5.1 Input Domain

The input domain of a program is contained by the set of all possible inputs to that program. This thesis is only concerned with branch coverage of a function, and thus uses the terms program and function interchangeably. For search–based algorithms the input domain constitutes the search–space. As stated, this includes all the global variables and formal parameters to a function containing the structure of interest, as well as the variables in a program that are externally assigned, *e.g.* via the `read` operation. Consider a program P with a corresponding input vector $P = \langle x_1, x_2, \ldots, x_n \rangle$, and let the domain of each input be $\langle D_1, D_2, \ldots, D_n \rangle$, such that $x_1 \in D_1, x_2 \in D_2$ and so forth.

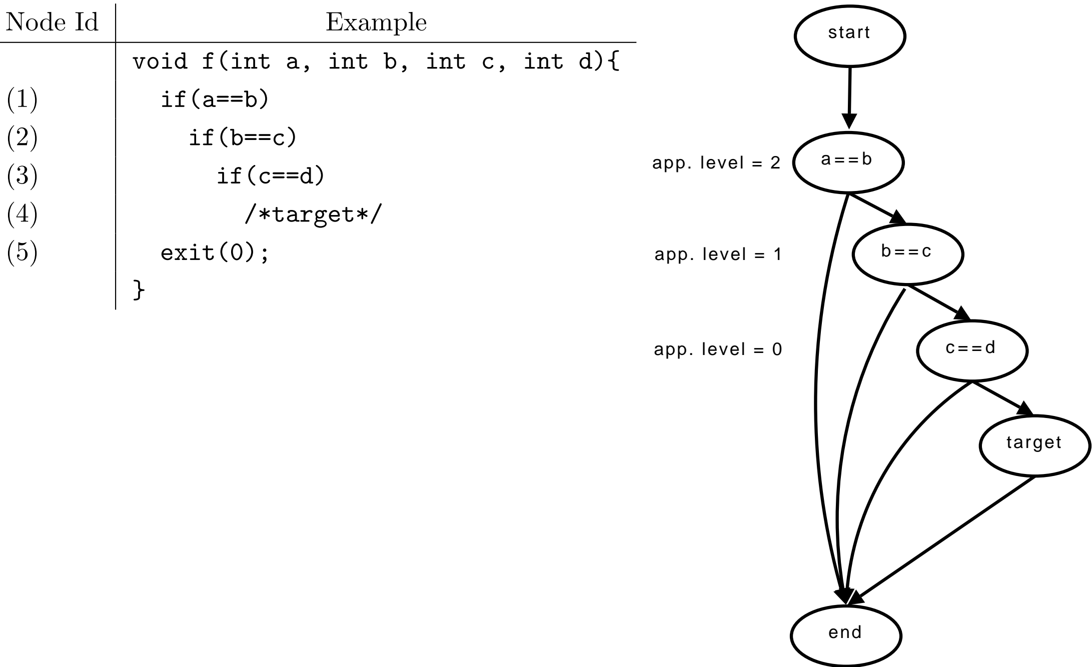
The domain D of a function can then be expressed as the cross product of the domains of each input: $D = D_1 \times D_2 \times \ldots \times D_n$.

1.5.2 Control and Data Dependence

Most dynamic test data generation techniques, as well as static analysis techniques such as symbolic execution, are based on either control flow graphs, data flow information, or both. A CFG is a directed graph $G = \langle N, E, n_s, n_e \rangle$, where N is a set of nodes, E a set of edges ($E \subseteq NxN$), $n_s \in N$ a unique start node and $n_e \in N$ a unique exit node. CFG's are used to represent the paths through a program, module or function. Each node $n \in N$ may represent a statement or a block of statements with no change in control flow. A CFG contains a single edge for each pair of nodes, (n_i, n_j) where control passes from one node, n_i, to another, n_j. Additionally E contains an edge (n_s, n_i) from the start node to the first node representing a statement or block, and at least one edge (n_i, n_e), $\{n_i \in N | n_i \neq n_e\}$, to the unique exit node.

CFG's can be used to extract *control dependence* information about nodes. To understand the notion of control dependence one first needs to clarify the concept of domination. A node n_i in a CFG is said to be *post–dominated* by the node n_j if every directed path from n_i to n_e passes through n_j (excluding n_i and the exit node n_e). A node n_j is said to be *control dependent* on n_i if, and only if there exists a directed path p from n_i to n_j and all nodes along p (excluding n_i and n_j) are post–dominated by n_j, and further n_i is not post–dominated by n_j.

The right column in Figure 1.1 contains the CFG for the code in the left column. Nodes 1, 2 and 3 are *branching nodes*; nodes which contain two or more outgoing edges (*branches*). Branching nodes correspond to condition statements, such as loop conditions, `if` and `switch` statements. The execution flow at these statements depends on the evaluation of the condition. In structural testing, these conditions are referred to as *branch predicates*. They provide search algorithms with the *branch distance* measure (see annotation of the CFG in Figure 1.1). The branch distance indicates how close the execution of a program comes to satisfying a branch predicate with a desired outcome. It is computed via the variables and their relational operators appearing in the predicates. Concrete examples of the branch distance and its use in structural testing are provided in Section 2.1.3.

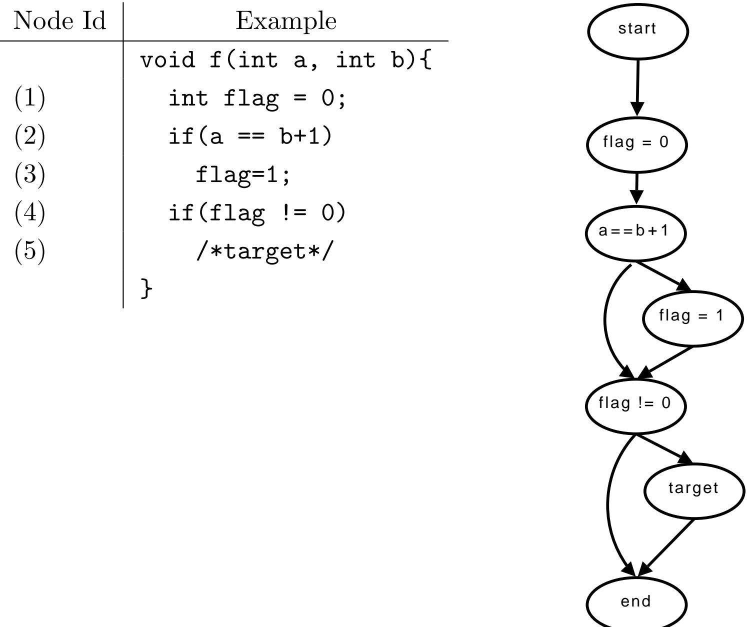


Node Id	Example
	`void f(int a, int b, int c, int d){`
(1)	`  if(a==b)`
(2)	`    if(b==c)`
(3)	`      if(c==d)`
(4)	`        /*target*/`
(5)	`  exit(0);`
	`}`

Figure 1.1: Code template in the left column is used to illustrate search–based test data generation. In the right column is the CFG of the code in the left column. The CFG is annotated with the *approach level* for node 4.

Data flow graphs express data dependencies between different parts of a program, module or function. Typically, data flow analysis concerns itself with paths (or sub–paths) from variable definitions to their uses. A variable is defined if it is declared or appears on the left hand side of an assignment operator. More generally, any operation which changes the value of a variable is classed as a definition of that variable.

The use of a variable can either be computational, *e.g.* the variable appearing on the right hand side of an assignment operation or as an array index, or a predicate use, with the variable being used in the evaluation of a condition. Data flow graphs capture information not explicit in CFGs. Consider the example shown in Figure 1.2. Node 5 is control dependent on node 4, which in turn is not control dependent on any node in the CFG except the start node. However, it is data dependent on node 3, because `flag` is defined at node 3 and there exists a *definition-clear* path from node 3 to node 5. Informally, any path from the definition of a variable x to its use can be considered a definition–clear path for x, if, and only if, the path does not alter or update the value of x. All paths are definition–clear with respect to the variables `a` and `b` in the example in Figure 1.2 .



Node Id	Example
	`void f(int a, int b){`
(1)	`  int flag = 0;`
(2)	`  if(a == b+1)`
(3)	`    flag=1;`
(4)	`  if(flag != 0)`
(5)	`    /*target*/`
	`}`

Figure 1.2: A code example with the corresponding CFG on the right illustrating how data flow information is not captured explicitly in a CFG. The true branch of node 4 is data dependent on node 3, but not control dependent on any other node except the start node.

Chapter 2

Literature Review

This chapter reviews work in the field of search–based automatic test data generation and dynamic symbolic execution based techniques. It focuses on the two most commonly used algorithms in structural search–based testing, genetic algorithms and hill climbing. Particular attention is paid to a local search method called 'alternating variable method', first introduced by Korel (Kor90), that forms the basis for most of the work in this thesis. This is followed by a discussion of techniques used for structural testing that are derived from static analysis. The chapter again only focuses on the two predominant techniques used in literature, symbolic execution and dynamic symbolic execution.

2.1 Search–Based Testing

The field of search–based testing began in 1976 with the work of Miller and Spooner (MS76), who applied numerical maximization techniques to generate floating point test data for paths. Their approach extracted a straight-line version of a program by fixing all integer inputs (in effect pruning the number of possible paths remaining) and further replacing all conditions involving floating point comparisons with path constraints of the form $c_i > 0$, $c_i = 0$ and $c_i \geq 0$, with $i = 1, ..., n$. These constraints are a measure of how close a test case is to traversing the desired path. For example, for a conditional i of the form `if(a != b)`, c_i corresponds to $abs(a - b) > 0$. Continuous real–valued functions were then used to optimize the constraints, which are negative when a test case 'misses' the target path, and positive otherwise. A test case satisfying all con-

Table 2.1: Branch functions for relational predicates introduced by Korel (Kor90).

Relational Predicate	Fitness Function	Rel
$a = b$	$abs(a - b)$	$= 0$
$a \neq b$	$abs(a - b)$	< 0
$a > b$	$b - a$	< 0
$a \geq b$	$b - a$	≤ 0
$a < b$	$a - b$	< 0
$a \leq b$	$a - b$	≤ 0

straints such that $0 \leq \sum_{i=1}^{n} c_i$, is guaranteed to follow the path in the original program which corresponds to the straight-line version used to generate the test data.

More than two decades later Korel (Kor90) adapted the approach taken by Miller and Spooner to improve and automate various aspects of their work. Instead of extracting a straight–line version of a program, Korel instrumented the code under test. He also replaced the path constraints by a measure known as *branch distance.* In Miller and Spooners' approach, the distance measure is an accumulation of all the path constraints. The branch distance measure introduced by Korel is more specific. First, the program is executed with an arbitrary input vector. If execution follows the desired path, the test case is recorded. Otherwise, at the point where execution diverged away from the path, a branch distance is computed via a *branch distance* function. This function measures how close the execution came to traversing the alternate edge of a branching node in order to follow the target path. Different branch functions exist for various relational operators in predicates (see Table 2.1). A search algorithm is then used to find instantiations of the input parameters which preserve the successfully traversed sub–path of the desired path, while at the same time minimizing the branch distance, so execution can continue down the target path.

Shortly after the work of Korel, Xanthakis *et al.* (XES+92) also employed a path–oriented strategy in their work. They used a genetic algorithm to try and cover structures which were left uncovered by a random search. Similar to the approach by Miller and Spooner, a path is chosen and all the branch predicates are extracted along the target path. The search process then tries to satisfy all branch predicates at once, in order to force execution to follow the target path.

Until 1992, all dynamic white–box testing methods were based around program paths. Specifically, to cover a target branch, a path leading to that branch would have to be selected (in case more than one path leads to the target branch). This places an additional burden on the tester, especially for complex programs. Korel alleviated this requirement by introducing a goal–oriented testing strategy (Kor92). In this strategy the path leading to the target is (largely) irrelevant. The goal–oriented approach uses a program's control flow graph to determine *critical*, *semi–critical* and *irrelevant* branching nodes with respect to a goal. It can be used to achieve statement, branch and MC/DC coverage. When an input traverses an undesired edge of a critical node (*e.g.* the false branch of node 1 in Figure 1.1), it signals that the execution has taken a path which cannot lead to the goal. In this case a search algorithm is used to change the input parameters, causing them to force execution down the alternative branch at the point where execution diverged away from the goal. Execution of an undesired branch at a semi–critical branching node does not prevent an input from reaching a goal *per se*. Semi–critical nodes contain an edge (in the CFG) to the goal node, but they also contain an edge to a loop node. While the search will try and drive execution down the desired branch, an input may be required to iterate a number of times through the body of a loop before being able to reach the goal. Finally, irrelevant branching nodes do not control any edges leading to a goal. A branching node n_i with $(n_i, n_j)/(n_i, n_k)$ is irrelevant with respect to node n_m, if n_m is either reachable or unreachable from both n_j and n_k. For example the nodes 1, 2, 3, 4 in Figure 2.1 are irrelevant for reaching node 6, because node 6 is only control dependent on node 5.

Ferguson and Korel later extended the goal–oriented idea to sequences of sub–goals in the chaining approach (FK96). The chaining approach constructs sequences of nodes, sub–goals, which need to be traversed in order to reach the goal. For example, a predicate in a conditional might depend on an assignment statement earlier on in the program. This data dependency is not captured by a control flow graph, and hence the goal–oriented approach can offer no additional guidance to a search. This problem is particularly acute when flag variables are involved. Consider the example shown in Figure 1.2 and assume node 5 has been selected as goal. The branch distance function will only be applied at node 4 and will produce only two values, *e.g.* 0 or 1. These two values correspond to an input reaching the goal and an input missing the goal respectively. Thus a search algorithm deteriorates to a random search. The probability

Node Id	Example
	`void testme(int x, int c1, int c2, int c3) {`
(1)	`  if(condition1)`
(2)	`    x--;`
(3)	`  if(condition2)`
(4)	`    x++;`
(5)	`  if(condition3)`
(6)	`    x=x;`
	`}`

Figure 2.1: Example used to demonstrate Korel's goal oriented (Kor92) test data generation approach.

of randomly finding values for both `a` and `b` in order to execute the true branch of node 2 is however, relatively small. The chaining approach tries to overcome this limitation by analysing the data dependence of node 4. It is clear that node 3 contains a definition of `flag`, and thus node 3 is classified as a sub–goal. The search then applies the branch distance function to the predicate in node 2 with the goal of traversing the true branch. Once the search has found values for `a` and `b` which traverse the true branch at node 2, `flag` is set to 1, leading to the execution of the goal.

The field of search–based software testing continues to remain an active area of research. McMinn (McM04) provides a detailed survey of work on SBST until approximately 2004. Over time, branch coverage has emerged as the most commonly studied test adequacy criterion (RMB$^+$95; BJ01; LBW04; MH06; MMS01; MRZ06; PHP99; SBW01; XXN$^+$05) in SBST, largely due to the fact that it makes an excellent candidate for a fitness function (HC04). Thus many recent papers continue to consider search–based techniques for achieving branch adequate test sets (LHM08; PW08; SA08; WBZ$^+$08).

In addition, structural coverage criteria such as branch coverage (and related criteria such as MC/DC coverage) are mandatory for several safety critical software application industries, such as avionics and automotive industries (Rad92). Furthermore, statement and branch coverage are widely used in industry to denote minimal levels of adequacy for testing (Bri98).

Much of the previous work considered imperative programming paradigms, but

the search–based approach has also been applied to problems of coverage for Object Oriented programming styles (AY08; CK06; SAY07; Ton04; Wap08).

The most popular search technique applied to structural testing problems has been the genetic algorithm. However, other search–based algorithms have also been applied, including parallel evolutionary algorithms (AC08), Evolution Strategies (AC05), Estimation of Distribution Algorithms (SAY07), Scatter Search (BTDD07; Sag07), Particle Swarm Optimization (LI08; WWW07) and Tabu Search (DTBD08). Korel (Kor90) was one of the first to propose a local search technique, known as the alternating variable method. Recent empirical and theoretical studies have shown that this search technique can be a very effective and highly efficient approach for finding branch adequate test data (HM09).

The remainder of this section provides a brief introduction to genetic algorithms before delving into a more detailed description of the alternating variable method. The section concludes by discussing fitness functions for the branch coverage adequacy criterion and their role in search–based testing.

2.1.1 Genetic Algorithms

Genetic algorithms first emerged as early as the late 1950's and early 1960's, primarily as a result of evolutionary biologists looking to model natural evolution. The use of GAs soon spread to other problem domains, leading amongst other things, to the emergence of *evolution strategies*, spearheaded by Ingo Rechenberg. Evolution strategies are based on the concept of evolution, albeit without the 'typical' genetic operators like crossover. Today's understanding of genetic algorithms is based on the concept introduced by Holland in 1975. Holland was the first to propose the combination of two genetic operators: crossover and mutation. The use of such genetic operators placed a great importance on choosing a good encoding for candidate solutions to avoid destructive operations.

Originally candidate solutions (phenotypes) were represented as binary strings, known as chromosomes. Consider the code example from Figure 1.1 and assume that the phenotype representation of a candidate solution is $\langle 3, 4, 8, 5 \rangle$, where each element in the vector maps to the inputs `a`, `b`, `c` and `d` respectively. The chromosome is formed by concatenating the binary string representations for each element in the vector, *i.e.*

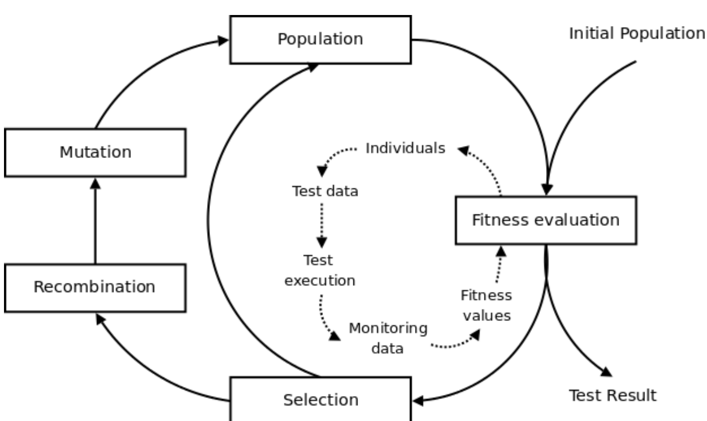


Figure 2.2: A typical evolutionary algorithm for testing.

11 100 1000 101. The optimization, or search process, then works on the genotypes by applying genetic operators to the chromosomes.

A binary representation may be unsuitable for many applications. Other representation forms such as *real value* (Wri90), *grey-code* (CS89) and *dynamic parameter encoding* (SB92) have been shown to be more suitable in various instances.

A standard GA cycle consists of 5 stages, depicted in Figure 2.2. First a population is formed, usually with random guesses. Starting with randomly generated individuals results in a spread of solutions ranging in fitness because they are scattered around the search–space. This is equal to sampling different regions of the search–space and provides the optimization process with a diverse set of 'building blocks'. Next, each individual in the population is evaluated by calculating its fitness via a fitness function Φ. The principle idea of a GA is that *fit* individuals survive over time and form even fitter individuals in future generations. This is an analogy to the 'survival of the fittest' concept in natural evolution where fit specimen have a greater chance of reproduction.

In a GA, the *selection operator* is used to pick individuals from a population for the reproduction process. This operator is problem and solution independent, but can have a great impact on the performance and convergence of a GA, *i.e.* the probability and time taken for a population to contain a solution. Individuals are either selected on the basis of their fitness value obtained by Φ, or, more commonly, using a probability assigned to them, based on their fitness value. This probability is also referred to as *selection pressure*. It is a measure of the probability of the best individual being selected from a population, compared to the average probability of selection of all individuals.

Desired String:	"Hello	World"		
	Crossover Point			
Candidate String 1:	"Hello	Paul"		Offspring 1: "Hello World"
		⇕	⇒	
Candidate String 2:	"This	World"		Offspring 2: "This Paul"

Figure 2.3: Example illustrating how the crossover operator in a GA combines parts of two candidate solutions, *Candidate String 1* and *Candidate String 2*, to form an offspring representing the desired string.

Once a pair of individuals has been selected, a *crossover operator* is applied during the recombination stage of the GA cycle. The crossover operation aims to combine good parts from each individual to form even better individuals, called *offspring* (see Figure 2.3). During the initial generations of a GA cycle this operator greatly relies on the diversity within a population.

While the crossover operator can suffice to find a solution if the initial population contains enough diversity, in many cases crossover alone will lead to a premature stagnation of the search. To prevent this, and have a greater chance of escaping *local optima*, an additional *mutation operator* is required. One of the principle ideas behind mutation is that it introduces new information into the gene–pool (population), helping the search find a solution. Additionally, mutation maintains a certain level of diversity within a population. The mutation operator is applied to each offspring according to a *mutation probability*. This measure defines the probability of the different parts of a chromosome being mutated. In a bit string for example, a typical mutation is to flip a bit, *i.e.* convert a 0 to 1 and vice versa. The importance of a mutation operator is best illustrated with an example. Consider the two chromosomes 0001 and 0000, and suppose the desired solution is 1001. No crossover operation will be able to generate the desired solution because crucial information has been lost in the two chromosomes. The mutation operator has a chance of restoring the required information (by flipping the first bit in a chromosome) and thus producing the solution.

Once offspring have been evaluated and assigned a fitness value, they need to be reinserted into the population to complete the reproduction stages. The literature distinguishes between two types of reproduction: *generational* and *steady state* (Sys89).

In the first, there is no limit on the number of offspring produced, and hence the entire population is replaced by a combination of offspring and parents to form a new generation. Steady state GAs introduce at most two new offspring into the population per generation (Sys91). Different schemes exist to control the size of the population, as well as choosing which members are to be replaced by offspring. In an *elitist* strategy for example, the worst members of a generation are replaced, ensuring fit individuals will always be carried across to the next generation. Other strategies include replacing a member at random or replacing members based on their inverse selection probability.

2.1.2 Hill Climb

Hill climbing is a well–known local search method, also classed as a *neighbourhood* search. In search algorithms the term neighbourhood describes a set of individuals (candidate solutions) that share certain properties. Consider an integer variable x, and assume $x = 5$. One way of defining the neighbours for x is to take the integers adjacent to x, *i.e.*, $\{4, 6\}$.

Typically a hill climber starts off at a randomly chosen point in the search–space (the space of all possible individuals). The search then explores the neighbourhood of the current individual, looking for better neighbours. When a better neighbour is found, the search moves to this new point in the search–space and continues to explore the neighbourhood around the new individual. In this way the search moves from neighbour to neighbour until an individual is no longer surrounded by a fitter neighbour than itself. At this point the search has either reached a local optimum or, indeed the global optimum.

Hill climbing comes in various flavours, such as simple ascent, steepest ascent (descent for minimization functions) and stochastic hill climb. In a simple hill climb strategy, the search moves to the first neighbour which is better than the current individual. In a steepest ascent strategy, all neighbours are evaluated and the search moves to the best overall neighbour. A stochastic hill climber lies somewhere in between; the neighbour moved to is chosen at random from all the possible candidates with a higher or equal fitness value.

The AVM alluded to in Section 2.1 works by continuously changing each numerical element of an individual in isolation. For the purpose of branch coverage testing, each element corresponds to an input to the function under test. First, a vector containing

the numerical type inputs (*e.g.* integers, floats) to the function under test is constructed. All variables in the vector are initialized with random values. Then so called *exploratory moves* are made for each element in turn. These consist of adding and subtracting a delta from the value of an element. For integral types the delta starts off at 1, *i.e.* the smallest increment (decrement). When a change leads to an improved fitness (of the candidate solution), the search tries to accelerate towards an optimum by increasing the amount being added (subtracted) with every move. These are known as *pattern moves.* The formula used to calculate the amount which is added or subtracted from an element is: $\delta = s^{it} * dir * 10^{-acc_i}$, where s is the *repeat base* (2 throughout this thesis) and it the repeat iteration of the current move, dir either -1 or 1, and acc_i the accuracy of the ith input variable. The accuracy applies to floating point variables only (*i.e.* it is 0 for integral types). It denotes a scale factor for any delta added or subtracted from an element. For example, setting the accuracy acc_i for an input to 1 limits the smallest possible move for that variable to adding a delta of ± 0.1. Increasing the accuracy to 2 limits the smallest possible move to a delta of ± 0.01, and so forth.

When no further improvements can be found for an element, the search continues exploring the next element in the vector. Once the entire input vector has been exhausted, the search recommences with the first element if necessary. In case the search stagnates, *i.e.* no move leads to an improvement, the search restarts at another randomly chosen location in the search–space. This is known as a *random restart* strategy and is designed to overcome local optima and enable the hill climber to explore a wider region of the input domain for the function under test.

2.1.3 Fitness Functions

Fitness functions are a fundamental part of any search algorithm. They provide the means to evaluate individuals, thus allowing a search to move towards better individuals in the hope of finding a solution. In the context of coverage testing, past literature has proposed fitness functions which can be divided into two categories: *coverage* and *control* based. In the former, fitness functions aim to maximize coverage, while in the latter they are designed to aid the search in covering certain control constructs. This section provides a brief treatment of the two approaches.

Coverage oriented approaches often provide little guidance to the search. They tend to reward (Rop97) or penalize (Wat95) test cases on the basis of what part of a

program has been covered or left uncovered respectively. As a result, the search is often driven towards a few long, easily executed program paths. Complex code constructs, *e.g.* deeply nested conditionals, or predicates whose outcome depends on relatively few input values from a large domain, as is often the case with flags, are likely to be only covered by chance.

Structure oriented approaches tend to be more targeted. The search algorithm will systematically try and cover all structures within a program required by the test adequacy criterion. The information used to guide the search is either based on branch distance (see Korel (Kor90)), control oriented (see Jones *et al.* (JSE96)), or a combination of both (see Tracey (Tra00) and Wegener *et al.* (WBS01)).

In a control oriented strategy, control flow and control dependence graphs are used to guide the search. For example Pargas *et al.* (PHP99) use critical branching nodes to evaluate individuals. The idea is that the more of these nodes a test case evaluates, the closer it gets to the target. A target branch or statement is control dependent on its critical branching nodes. The resulting fitness landscape for such an approach is likely to be coarse grained and contain plateaus, because the search is unaware of how close a test case was to traversing the desired edge of a critical branching node. Take the example in Figure 1.1 and assume node 4 has been selected as target. Node 4 is control dependent on node 3, which in turn is control dependent on node 2, which is control dependent on node 1. All test cases following the false branch of node 1 will have identical fitness values, *i.e.* 3. The same is true for any test case traversing the false branch at node 2; they will all have a fitness value of 2.

A combined strategy uses both, the branch distance as well as control dependence information. Tracey (TCMM00) was the first to introduce a combination of the two. Whenever execution takes an undesired edge of a critical branching node, *i.e.* it follows a path which cannot lead to the target, a distance is calculated, measuring how close it came to traversing the alternate edge of the branching node. However, unlike in Korel's approach, the distance measure is combined with a ratio expressing how close a test case came to traversing a target with respect to its critical branching nodes. The template for the fitness function used by Tracey is:

$$\frac{executed}{dependent} \times branch_distance$$

Table 2.2: Branch distance measure introduced by Tracey (Tra00) for relational predicates. The value K is a failure constant which is always added if a term is false.

Relational Predicate	Branch Distance Formulae
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 $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$\neg a$	Negation is moved inwards and propagated over a

Table 2.3: Objective functions introduced by Tracey (Tra00) for conditionals containing logical connectives. z is one of the objective functions from Table 2.2.

Logical Connectives	Value
$a \wedge b$	$z(a) + z(b)$
$a \vee b$	$min(z(a), z(b))$
$a \Rightarrow b$	$min(z(\neg a), z(b))$
$a \Leftrightarrow b$	$min((z(a) + z(b)), (z(\neg a) + z(\neg b)))$
$a\ xor\ b$	$min((z(a) + z(\neg b)), (z(\neg a) + z(b)))$

dependent is the number of critical branching nodes for a target, and *executed* is the number of critical branching nodes where a test case traversed its desired edge. The formulae used for obtaining *branch_distance* for single predicates as well as predicates joined by logical connectives are shown in Tables 2.2 and 2.3 respectively. Combining the distance measure with a ratio of executed critical branching nodes in effect scales the overall fitness value. Take the example from Figure 1.1 again. Node 4 has 3 critical branching nodes: 1, 2 and 3. Suppose the example function is executed with the input vector $\langle 3, 6, 9, 4 \rangle$. This vector will take the false branch of node 1. Using Tracey's fitness function shown above (assuming $K = 1$), will yield the following values:

branch_distance	4
executed	1
dependent	3
fitness	1.33

While in many scenarios the work of Tracey *et al.* can offer an improved fitness landscape compared to the formulae used by Pargas *et al.*, the fitness calculations may still lead to unnecessary local optima (McM05).

Wegener *et al.* (WBP02; WBS01) proposed a new fitness function to eliminate this problem, based on *branch distance* and *approach level*. The approach level records how many of the branch's control dependent nodes were not executed by a particular input. The fewer control dependent nodes executed, the 'further away' an input is from executing the branch in control flow terms. Thus, for executing the true branch of node 3 in Figure 1.1, the approach level is:

- 2 when an input executes the false branch of node 1;
- 1, when the true branch of node 1 is executed followed by the false branch of node 2;
- zero if node 3 is reached.

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current 'target' branch. Taking the true branch from node 3 as an example again, if the false branch is taken at node 1, the branch distance is computed using $|a-b|$, whilst $|b-c|$ is optimized if node 2 is reached but executed as false, and so on. The branch distance is normalized to lie between zero and one, and then added to the approach level.

2.2 Static Analysis Based Testing

Static analysis techniques do not require a system under test to be executed. Instead test cases can be obtained by solving mathematical expressions or reasoning about properties of a program or system. One of the main criticisms of static methods is the computational cost associated with them. However, static analysis techniques can be used to prove the absence of certain types of errors, while software testing may only be used to prove the presence of errors. This often makes static analysis based techniques indispensable, especially in safety critical systems.

The remainder of this section discusses symbolic execution, one of the first static analysis techniques applied to testing, and its successor, dynamic symbolic execution.

2.2.1 Symbolic Execution

Symbolic execution is a source code analysis technique in which program inputs are represented as symbols and program outputs are expressed as mathematical expressions involving these symbols (The91). Symbolic execution can be viewed as a mix between testing and formal methods; a testing technique that provides the ability to reason about its results, either formally or informally (Kin76). For example, symbolic execution of all paths creates an execution tree for a program (also known as execution model). It is a directed graph whose nodes represent symbolic states (CPDGP01). This model can be used to formally reason about a program, either with the aid of a path description language (CPDGP01), or in combination with explicit state model checking (Cat05).

During symbolic execution, a program is executed statically using a set of symbols instead of dynamically with instantiations of input parameters. The execution can be based on forward or backward analysis (JM81). During backward analysis, the execution starts at the exit node of a CFG, whereas forward analysis starts at the start node of a CFG. Both type of analyses produce the same execution tree, however forward analysis allows faster detection of infeasible paths.

Symbolic execution uses a path condition (*pc*) to describe the interdependency of input parameters of a program along a specific path through a program. It consists of a combination of algebraic expressions and conditional operators. Any instantiation of input parameters that satisfies the *pc* will thus follow the path described by the *pc*. In the absence of preconditions, *pc* is always initialized to *true*, *i.e.* no assumptions about the execution flow are made.

Suppose the path leading to node 4 in Figure 1.1 is selected as target, and the inputs to the function are represented by the symbols a_0, b_0, c_0 and d_0. During symbolic execution the path condition describing this path would be built as follows. Initially it is set to *true* as previously mentioned. After executing the true branch of node 1, it is updated to $\langle a_0 = b_0 \rangle$, to capture the condition necessary for any concrete instantiations of a_0 and b_0 to also traverse the true branch of node 1. At the decision node 2, the path condition is updated to $\langle a_0 = b_0 \wedge b_0 = c_0 \rangle$, and finally to $\langle a_0 = b_0 \wedge b_0 = c_0 \wedge c_0 = d_0 \rangle$ at node 3. Once the path condition has been constructed, it can be passed to a constraint solver. If the constraint solver determines that no solution exists, it follows that the

path described by the path condition is *infeasible*. Indeed, one of many strengths of symbolic execution is the ability to detect infeasible paths. Otherwise, the solution returned by a constraint solver can be used to execute the program along the path described by the path condition.

Early implementations of symbolic execution were often interactive. For example, they would require the user to specify which path to take at forking nodes (Kin76). Without this interaction, symbolic execution will try and explore all program paths. However, this is infeasible in the presence of loops and recursion, because symbolic execution systematically unfolds both. Further, even medium sized programs often suffer from path explosion which makes symbolic execution so computationally expensive that it may become infeasible in practice.

Constraint solvers are the 'Achilles heel' of symbolic execution because they often cannot handle non–linear constraints or constraints involving floating point variables. Without constraint solvers being able to support all the constraints in a path condition, symbolic execution is useless for testing. Sometimes the nature of a program inherently *prevents* reasoning about a constraint, *e.g.* consider the code below.

```
int testme(int x, int y){
  if(hash(x) == y)
    return 1;
  return 0;
}
```

If `hash` is a hash or cryptographic function, it has been mathematically designed to make reasoning computationally infeasible (GdHN+08). Tillmann and Schulte (TS06) proposed the use of *symbolic mock objects* to alleviate some limitations on constraint solvers. However, the use of such mock objects may not always be feasible in practice.

Another problem for constraint solvers are floating point computations. Reasoning about floating point expressions must not be based on common mathematical principles, but instead must obey the machine dependent semantics of floating point computations. Take the example in Figure 2.4. A constraint solver based on real numbers (or rational numbers) may wrongly flag the path $\langle 1, 2, 3, 4, 5 \rangle$ as infeasible. Yet with `x` taking on any $IEEE - 754$ single–format floating point number of the closed interval $[1.401298464324817e - 45, 32767.9990234]$, the path is traversed (BGM06).

Node Id	Example
	`void f(float x){`
(1)	`  float y=1.0e12, z;`
(2)	`  if(x > 0.0)`
(3)	`    z = x + y;`
(4)	`  if(z == y)`
(5)	`    /*target*/`
	`}`

Figure 2.4: Example used to illustrate how machine semantics differ from mathematical principles over reals and rational numbers. Arithmetic suggests that the path $\langle 1, 2, 3, 4, 5\rangle$ is infeasible, when in practice there exist values for x that satisfy this path.

Work has been done to address the floating point problem in symbolic execution (BGM06) though the proposed technique has not yet been adapted by popular constraint solvers (dMB08; DdM06).

2.2.2 Dynamic Symbolic Execution

Dynamic symbolic execution (GKS05; CE05; SMA05; TdH08) builds on the ideas of symbolic execution. For a given path through a program, symbolic execution involves constructing a system of constraints in terms of the input variables that describe when the path will be executed. As mentioned in the previous section, such a path condition can easily become unsolvable if it contains expressions that cannot be handled by constraint solvers. This is often the case with floating–point variables, or non–linear constraints to name but a few examples. Dynamic symbolic execution alleviates some of the problems by combining concrete execution with symbolic execution. The idea is to simplify a path condition by substituting sub–expressions with runtime values, obtained through dynamic executions of a program. This substitution process can remove some of the non–linear sub–expressions in a path condition, making them amenable to a constraint solver.

Suppose the function in Figure 2.5 is executed with the random values `536` and `156` for `x` and `y` respectively. The path taking the false branch is executed. The path condition is $\langle x_0 * y_0 \geq 100\rangle$, where x_0 and y_0 refer to the symbolic values of the input variables `x` and `y` respectively. Suppose you want to execute the true branch instead, *i.e.*

```
void testme2(int x, int y)
{
    if (x * y < 100)
        // ...
}
```

Figure 2.5: Example for demonstrating dynamic symbolic execution. The branch predicate is non–linear.

find values for `x` and `y` which satisfy the following path condition: $\langle x_0 * y_0 < 100 \rangle$. Since this path condition contains non–linear constraints, one strategy in dynamic symbolic execution is to replace x_0 with its concrete value, `536` (SMA05). The path condition becomes $\langle 536 * y_0 < 100 \rangle$, which is now linear and can be passed to the constraint solver to find an appropriate value for `y` (*i.e.* zero or any negative value).

In order to explore different paths through a program, dynamic symbolic execution based techniques first execute the program with an arbitrary input vector. The corresponding path condition forms the basis for successive iterations of the test data generation process. A path condition describes one succinct execution path through a program, thus a different path condition, if feasible, will describe another succinct path through the same program. Recall the example from Figure 1.1 and suppose the function is executed with the values `0`, `1`, `2`, and `3` for `a`, `b`, `c`, and `d` respectively. The path condition is $\langle a_0 \neq b_0 \rangle$. One possible strategy is to invert the last constraint in a path condition, *e.g.* similar to performing a depth first search. The new path condition is $\langle a_0 = b_0 \rangle$, which is passed to a constraint solver. Assume the constraint solver returns the value `0` for both `a` and `b`. Executing the function with the updated values `0`, `0`, `2`, and `3` for `a`, `b`, `c`, and `d` yields the path condition $\langle a_0 = b_0 \wedge b_0 \neq c_0 \rangle$. Following the same principle, the last constraint is inverted again to give $\langle a_0 = b_0 \wedge b_0 = c_0 \rangle$, which is passed to the constraint solver. Suppose the constraint solver now returns the value `0` for `a`, `b` and `c`. The function is executed once more, this time with the values `0`, `0`, `0`, and `3` for `a`, `b`, `c`, and `d`. This process continues until all feasible execution paths through a function (or unit in the case of interprocedural testing) have been explored.

One of the first dynamic symbolic execution based tools to emerge was developed by Godefroid *et al.* (GKS05) during their work on directed random testing and the DART tool. DART does not attempt to solve constraints involving memory locations,

i.e. pointer inputs to a program. Instead, pointer inputs are randomly initialized to either the constant *null* or a new memory location. As a consequence DART is not ideally suited to test complex data structures which may require the generation of cycles, *e.g.* as part of a linked list implementation. Another limitation to DART are non–linear constraints because they fall outside the scope of DART's constraint solver. To overcome this problem DART replaces non–linear expressions with their runtime value similar to the example given at the start of this section. Unlike in the example, DART replaces the entire sub–expression, not just a single multiplicand. Therefore DART deteriorates to a pure random search for the example in Figure 2.5.

Cadar and Engler independently developed a method called Execution Generated Testing (EGT) (CE05) around the same time as DART. EGT starts with pure symbolic execution. When constraints on a programs input parameters become too complex, symbolic execution is paused and the path condition collected thus far is instantiated with concrete inputs. Runtime values are then used to simplify symbolic expressions so that symbolic execution can continue with a mix of symbolic variables and constants. Sen *et al.* further developed the work on DART in their tool called CUTE (SMA05). CUTE extends DART by including a custom solver for constraints over memory locations and also offers a more refined approach to handling non–linear constraints. Instead of replacing an entire expression, as is the case in DART, CUTE only replace one multiplicand with its concrete value, while maintaining a symbolic state for the other. The first path explored by the CUTE tool is the path executed where inputs of primitive type are zero (or of a random value, depending on the settings used). If the function involves pointer variables, these are always initially set to *null*. However, further paths through the program may require pointers to point to a specific memory location, *e.g.* when representing a data structure. In order to find the 'shape' of such a data structure, CUTE incorporates symbolic variables for pointers in the path condition. A graph–based process is used to check that the constraints over the pointer variables are feasible, and finally, a simple procedure is used to actually build the data structure required.

For the program of Figure 2.6(a), and the path that executes the true branch at each decision, CUTE accumulates the path constraint $\langle ptr_0 \neq NULL \wedge left_0 \neq NULL \wedge right_1 = ptr_0 \rangle$. CUTE keeps a map of which symbolic variable corresponds to which point in the data structure, for example, $left_0$ maps to `ptr->left`. The feasibility

Node Id	Example
	`void testme4(item* ptr)`
	`{`
(1)	`    if (ptr != NULL)`
(2)	`        if (ptr->left != NULL)`
(3)	`            if (ptr->left->right == ptr)`
(4)	`                // ...`
	`}`

(a) Code snippet

(b) CUTE feasibility graph for path which executes all decisions in 2.6(a) as true

Figure 2.6: Example for demonstrating pointer handling in CUTE.

check involves the construction of an undirected graph, which is built incrementally at the same time as the path condition is constructed from the conditions appearing in the program. The nodes of the graph represent abstract pointer locations, with node labels representing the set of pointers which point to those locations. A special node is initially created to represent *null*. Edges between nodes represent inequalities. After statement 1 in the example, the graph consists of a node for ptr_0 with an edge leading to the *null* node. When statement 2 is encountered, a new node is constructed for $left_0$, with an edge to *null*. Finally, $right_1$ is merged into the existing ptr_0 node, as they must point to the same location (Figure 2.6(b)). Feasibility is checked as each constraint is added for each decision statement. An equality constraint between two pointers `p` and `q` is feasible if, and only if, there is no edge in the graph between nodes representing the locations of `p` and `q`. An inequality constraint between `p` and `q` is feasible if, and only if, the locations of `p` and `q` are not represented by the same node.

If the path condition is feasible, the data structure is built incrementally. Each new branching decision adds a new constraint to the path condition, and the data structure is created on the basis of each constraint using the rules of Table 2.4.

CREST (BS08) is a recent open source successor to CUTE. Its main difference to

Table 2.4: Dynamic data structure creation according to individual constraints encountered along the path condition for CUTE.

Constraint	CUTE
$m_0 = null$	Assign $null$ to m_0
$m_0 \neq null$	Allocate a new memory location pointed to by m_0
$m_0 = m_1$	Make m_1 alias m_0
$m_0 \neq m_1$	Allocate a new memory location pointed to by m_1

CUTE is a selection of different path exploration strategies. Most dynamic symbolic execution techniques (GKS05; CE05; SMA05) use a depth first, backtracking exploration strategy. With CREST a user can choose between depth first, random or control flow directed explorations. Burnim and Sen (BS08) show that a control flow directed path exploration can lead to higher branch coverage than any of the other strategies.

Pex (TdH08) is a parametrized unit testing framework developed by Microsoft. All dynamic symbolic execution tools for C instrument the source code of the program under test to perform symbolic execution. This has some obvious limitations: common library functions, such as those found in `stdlib.h`, remain uninstrumented and thus outside the scope of a tool's symbolic engine. Pex on the other hand performs its instrumentation at the .NET intermediate language level. It contains a complete symbolic interpreter for safe .NET code, *i.e.* code that does not use pointer arithmetic, and it can also reason about a limited number of unsafe features used in .NET. Its sophisticated symbolic interpreter enables Pex to construct path conditions which capture constraints over input variables more precisely.

Most publicly available dynamic symbolic execution tools test functions in isolation without pre– or post–conditions, which can lead to false alarms because a function may be executed with inputs which are infeasible in the functions 'real' calling context. Yet, testing large programs with an interprocedural testing tool may yield low coverage. Chakrabarti and Godefroid (CG06) propose to partition software into a number of smaller units to alleviate this problem. The test objective is to achieve full coverage of all the functions contained within a unit. This ensures functions are called only from within a valid calling context and thus have implicit preconditions attached.

The majority of literature on dynamic symbolic execution based testing has augmented static analysis techniques, primarily symbolic execution, with dynamic test data

generation techniques, specifically random testing. Inkumsah and Xie (IX07) were the first authors to propose a framework (EVACON) combining evolutionary testing with dynamic symbolic execution. Their framework targets test data generation for object oriented code written in JAVA. They use two existing tools, eToc (Ton04), an evolutionary test data generation tool for JAVA, and jCUTE, an explicit path model checker (SA06), based on the same principles as CUTE. The first is used to construct method sequences to test a JAVA method, while the second is used to optimize the coverage for the method under test. eToc uses a genetic algorithm to evolve method sequences along with their parameters as a means to achieve unit testing. The parameters used in the sequence of method calls are randomly initialized. One drawback of eToc is that new parameter values are only introduced into the population via the mutation operator, which randomly changes a parameter value within given bounds. This may make the search inefficient and even ineffective for certain branches because genetic operators need to share their effort between evolving sequences and evolving their parameters. However, eToc provides an interface allowing customized input parameter generation. The EVACON framework uses this feature to combine jCUTE with eToc. Unlike eToc, jCUTE provides a systematic approach to achieving branch coverage based on a combination of symbolic execution and random testing.

First, method call sequences (test cases) are constructed by eToc. These test cases are then passed to jCUTE, which will try and generate parameter values to cover all feasible branches for a particular sequence call construct. The sequences with their modified parameter values are then returned to eToc, to further evolve the order and number of method calls in a sequence.

Consider the example from Figure 2.7. One possible method sequence constructed by eToc for the method `m` might look like:

```
$a=objA():a.m(int) @ 8
```

This sequence will not reach any of the three targets labelled in Figure 2.7. However, jCUTE is able to optimize this call sequence by generating the parameter values `3` and `5` to cover the targets labelled `/*target 1*/` and `/*target 2*/` respectively. Note that in order to cover `/*target 3*/`, a new method call needs to be inserted into the sequence. jCUTE returns its test cases to eToc, which will evolve them further, in particular inserting or deleting method calls from the sequences, or changing their

```
public class objA{
 private int x;
 public void setX(int x)
 {
   this.x = x;
 }
 public void m(int x)
 {
   if(x == 3)
      /*target 1*/
   else if (x==5)
      /*target 2*/

   if(this.x == x)
     /*target 3*/
 }
}
```

Figure 2.7: Code example used to illustrate how the EVACON framework (IX07) is able to optimize both, the method call sequences required to test the function m, as well as the method parameters to achieve coverage of m.

order via crossover operations. Assume a new sequence, constructed by eToc, includes a call to `setX`, generating

```
$a=objA():a.setX(int):a.m(int) @ 8 5
```

This new test case is again passed to jCUTE, which will find instantiations for the primitive method parameters to cover the branch labelled `/*target 3*/`. The eToc – jCUTE cycle continues until a stopping criterion has been reached.

Inkumsah and Xie (IX07) evaluated their framework on a set of benchmark programs for white–box test data generation tools. They are able to report an improved coverage using the EVACON framework compared to the coverage achieved by the standalone use of either eToc or jCUTE.

Chapter 3

Augmented Search–Based Testing

3.1 Introduction

The last chapter described how meta–heuristic algorithms can be applied to the problem of automated structural test data generation. It provided a brief overview of the search–based software testing field, which continues to attract a lot of interest from the academic community. Yet despite this, there still is a lack of publicly available tools that provide researchers with facilities to perform search–based structural testing. This chapter introduces such a tool, AUSTIN [1]. Most previous work on SBST for structural coverage has used genetic algorithms for test data generation and has tended to be evaluated on relatively small scale systems, consisting of laboratory 'toy examples', such as the well–known triangle classification program. AUSTIN uses a variant of Korel's (Kor90) alternating variable method and augments it with techniques adapted from recent work on dynamic symbolic execution (GKS05; CE05; SMA05; TdH08). It can handle a large subset of C, though there are some limitations. Most notably AUSTIN cannot generate meaningful inputs for strings, void and function pointers, as well as union constructs. Variable argument length functions are also not supported. However, AUSTIN has been applied out of the box to real industrial code from the automotive industry (see Section 3.4) as well as a number of open source programs (see Chapter 4).

[1] The name is derived from AUgmented Search–based TestINg.

3. AUGMENTED SEARCH–BASED TESTING

To address the need for SBST to be evaluated on real–world examples, an empirical study was performed in which AUSTIN was compared against an evolutionary testing framework, which was developed as part of the EvoTest project (GKWV09). The framework represents a state–of–the–art evolutionary testing system and has been applied to case studies from the automotive and communications industry. Three case studies from the automotive industry, provided by Berner & Mattner Systemtechnik GmbH, formed the benchmark which AUSTIN was compared against for effectiveness and efficiency when generating branch adequate test data.

Automotive code was chosen as the benchmark for two reasons. Firstly, the use of these systems as experimental subjects goes some way towards addressing the current lack of previous empirical results concerning real–world examples. Secondly, the automotive industry is subject to testing standards that mandate structural coverage criteria and so the developers of production code for automotive systems are a natural target for automated test data generation techniques, such as those provided by AUSTIN.

The primary contributions of this chapter are as follows:

1. A fully featured search–based software testing tool for C.

2. A description of how features adapted from several strands of research on test data generation are incorporated into the design and implementation of the AUSTIN tool. These extend its applicability to a subset of C (not previously covered by any SBST tool) and also improve its effectiveness.

3. An empirical study that evaluates AUSTIN on automotive systems for branch coverage. The results support the claim that AUSTIN is both efficient and effective.

The rest of this chapter is organised as follows: Section 3.2 introduces AUSTIN and presents detailed descriptions of the different techniques implemented. Section 3.3 provides details about the ETF against which AUSTIN has been compared. The empirical study used to evaluate AUSTIN alongside the hypotheses tested, evaluation and threats to validity are presented in Section 3.4. Section 3.5 concludes the chapter.

3.2 AUSTIN

AUSTIN is a structural test data generation tool which combines a simple hill climber for integer and floating point type inputs with a set of constraint solving rules for pointer type inputs. It has been designed as a unit testing tool for C programs. AUSTIN considers a unit to be a function under test and all the functions reachable from within that function. It can be used to generate a set of input data for a given function which achieve branch coverage for that function. During the test data generation process, AUSTIN does not attempt to execute specific paths through a function in order to cover a target branch; the path taken up to a branch is an emergent property of the search process. The search is guided by an objective function that was introduced by Wegener *et al.* (WBS01) for the Daimler evolutionary testing system and explained in Section 2.1.3.

Similar to dynamic symbolic execution based testing (GKS05; SMA05; TdH08), AUSTIN also instruments the program under test to perform a pseudo–symbolic execution of the program along the concrete path of execution for a particular input vector. Collecting constraints over input variables via symbolic execution serves to aid AUSTIN in solving constraints over memory locations, denoted by pointer inputs to a function. Consider the example given in Figure 3.1 and suppose execution follows the false branch at node 1. AUSTIN will use a custom procedure to solve the constraint over the physical memory location denoted by the pointer variable `one`. On the other hand, if the false branch is taken at node 2, an AVM is used to satisfy the condition at node 2.

3.2.1 Floating Point Variables

Prior work based on the AVM (HM09; LMH09) required a tester to specify an *accuracy* for each floating point input to the function under test. The accuracy is used as a scale factor for any delta added or subtracted from an input. For example, setting the accuracy for floating point variables to 1 means scaling the delta added to an element by ± 0.1. Using a fixed size accuracy has some obvious limitations. Consider the two examples in Figure 3.3: for the function `testme1`, a scale factor of 0.1 does not prevent the search from quickly finding a solution. For the second function `testme2` however, the search is likely to fail if the scale factor remains fixed at 0.1. Suppose that the input

Node Id	Example
	`typedef struct item {`
	`   int key;`
	`};`
	`void testme(item* one) {`
(1)	`   if ( one != null ) {`
(2)	`      if ( one->key == 10 )`
	`         // target`
	`}`

Figure 3.1: Example C code used for demonstrating how AUSTIN combines custom constraint solving rules for pointer inputs with an AVM for other inputs to a function under test. The goal is to find an input which satisfies the condition at node 2.

parameter `d` in `testme2` from Figure 3.3 is initialized to the value `0.100000001490116`. Note that the last 7 digits are due to the inability to represent the value 0.1 accurately with a floating point variable. With a fixed scale factor of 0.1, the search adds or subtracts at most 0.100000001490116 from the current value of `d`. Thus the search has to partly rely on rounding errors to find a value for `d` that lies between `1.34876` and `1.34877`.

AUSTIN solves this problem by including the accuracy parameter in its optimization process. This extension has been implemented as part of the *neighbourhood_explored* procedure shown in Figure 3.2. Initially each floating point variable is optimized by scaling the amount added or subtracted to it by 0.1. Each floating point type (*i.e.* `float`, `double`) has a maximum accuracy associated with it. This limit is machine dependent, but commonly lies at 6 for single floating point types and 15 for `double` types. The limit presents a lower bound for scaling a delta.

Whenever the AVM gets stuck on a local optimum, AUSTIN tries to increase the accuracy of floating point variables before resorting to a random restart. If an increase in accuracy results in an improvement, AUSTIN restarts its exploratory moves and will repeat the modification to a variable's accuracy next time a local optimum is reached. When a change does not lead to further improvements, or the accuracy exceeds the limit of the floating point type associated with the variable (*e.g.* 6 or 15), the search moves on to the next (floating point) variable. Once all improving moves for a solution have been exhausted (including changing the accuracy of floating point inputs), AUSTIN

```
Global Inputs: input_index, increased_precision, precision_success, precision_index
neighbourhood_explored (s : candidate_solution)
 primitives := vector of all primitive type variables in s
 if input_index >= primitives.length then
   if precision_index >= primitives.length then
     return true
   end if
   if increased_precision and not(precision_success) then
     v := primitives.at(precision_index)
     if acc(v) > 1 then
       v := decrease acc of v by one
       update v in primitives
       s := (s with updated primitives)
     end if
     increment precision_index
   end if
   precision_success := false
   increased_precision := false
   while precision_index < primitives.length do
     v := primitives.at(precision_index)
     if typeOf(v) ≠ floating point type then
       increment precision_index
       continue
     else if typeOf(v) = single precision type then
       limit := 6 // limit is machine dependent
     else
       limit := 15 // limit is machine dependent
     end if
     if acc(v) >= limit then
       increment precision_index
     else
       increment accuracy of v by one
       update v in primitives
       s := (s with updated primitives)
       increased_precision := true
       reset_exploration_parameters()
       return false
     end if
   end while
   precision_index := 0
   return true
 else
   return false
 end if
```

Figure 3.2: Pseudo code illustrating the method for checking if the search is stuck at a local optimum. Part of this method also includes the steps for varying the accuracy of floating point variables as described in Section 3.2.1.

void testme1(float d)
{
  if(d > 1.34876)
    //target
}
	void testme2(float d)
	{
	  if(d > 1.34876 && d < 1.34877)
	    //target
	}

Figure 3.3: Example for demonstrating how AUSTIN optimizes floating point type inputs. Previous work (MH06; LMH09) commonly fixed the accuracy for floating point variables in the AVM to 1 or 2 decimal places. Such a strategy suffices to generate inputs to satisfy the branch predicate in the left column, but is unlikely to succeed for the branch predicate in the right column. AUSTIN is able to automatically satisfy the branch predicates in both the left and right columns.

performs a random restart. At every restart, the accuracy for each floating point variable is reset to 1.

Recall the example function `testme2` in Figure 3.3, and assume the formal parameter `d` is initially assigned 0. AUSTIN starts adding the values ± 0.1 to `d`. Note that due to the imprecision of floating point numbers, it will actually add a value similar to ± 0.100000001490116 because 0.1 cannot be represented accurately. Since $+0.100000001490116$ is closer to the desired value for `d`, *i.e.* closer to the interval $]1.34876, 1.34877[$, than -0.100000001490116, AUSTIN will continue to add an ever increasing amount to `d`. Eventually `d` will be greater than the upper bound of the desired interval (*i.e.* `1.34877`) and AUSTIN has to backtrack until it 'gets stuck', *i.e.* adding ± 0.100000001490116 does not lead to further improvements. At this point AUSTIN refines the scale factor for a given variable, *e.g.* by changing it from 0.1 to 0.01. Using this new scale factor, AUSTIN is able to get the value of `d` even closer to the desired interval. Next time AUSTIN 'gets stuck', it reduces the scale factor further from 0.01 to 0.001, and so forth. By continuing to adjust the scale factor, AUSTIN is able to find a value for `d` which lies within the desired interval.

3.2.2 Example

The previous section explained how the AVM has been extended to automatically optimize floating point variables without requiring human assistance. The basic AVM described in Section 2.1.2 has also been extended to add automatic support for pointers

and dynamic data structures. This extension is grounded in dynamic symbolic execution and is best explained with an example before proceeding with a more detailed description.

AUSTIN works by modelling each global and formal parameter of a function under test as a scalar symbolic variable. In the case of pointers, the target of a pointer (*i.e.* the memory pointed to by the pointer) is also represented through a scalar symbolic variable. Suppose the true branch of node 2 in Figure 3.1 is selected as the target. Let one_0 denote the symbolic variable for the the formal parameter of `testme`. Initially every pointer input to the function is assigned the constant *null*. The path condition describing the execution of `testme` with this input is $\langle one_0 = null \rangle$. To bring the search closer to the target branch, the constraint of the last critical branching node where execution diverged away from the target (node 1) is inverted. AUSTIN solves the constraint by assigning a new memory location (via `malloc`) large enough to hold an object of type `item` to the input `one`. Intuitively the effect of this operation is that of extending the list of formal parameters for `testme` with the members of the data structure `item`. Let the additional input parameter `key` be denoted by the symbol one_key_0. In its default mode, AUSTIN sets `one->key` to 0. The program is executed with the updated inputs, and the new path constraint describing the execution is $\langle one_0 \neq null \wedge one_key_0 \neq 10 \rangle$. This time the critical branching node does not contain constraints over physical memory locations, and instead involves a comparison between two integer types. AUSTIN thus uses the AVM described in Section 2.1.2 to find values for `one->key` that satisfy the condition at node 2. It will start by trying the values $-1, +1$, before continuing to increase `one->key` to $3, 7$ and finally 15. At this point the minimum (`one->key == 10`) has been overshot, so the AVM repeats the exploratory–pattern move cycle for as long as necessary until the minimum is reached. When `one->key == 10`, the target branch of node 2 is executed.

3.2.3 Code Preparation

AUSTIN makes extensive use of the CIL (NMRW02) framework and its API for tasks ranging from code instrumentation and control dependence analysis to the symbolic evaluation described in Section 3.2.4. CIL is also used to transform and simplify source code prior to the test data generation process. The following CIL and custom transformations are used:

1. **Prepare Control Flow Graph:** This CIL option (`--domakeCFG`) converts all `break`, `switch`, `default` and `continue` statements and labels into equivalent `if` and `goto` constructs. This simplifies AUSTIN's control dependence analysis and also ensures all branching nodes in the CFG denote an `if` statement.

2. **Three–Address Code:** This CIL option (`--dosimplify`) transforms the source code into a simpler three–address code. This transformation introduces well typed temporary variables, ensuring all statements are side effect free and also that branching nodes only contain constraints over equal type variables.

3. **One Instruction Per Statement:** This custom transformation splits a list of instructions, *i.e.* a sequence of statements where flow of control implicitly falls through, into a sequence of block statements. This ensures that AUSTIN can insert its instrumentation of the source code in the correct place for every instruction.

4. **Remove Storage:** This custom transformation removes all `register` storage specifiers. Variables stored in a register do not have an address and hence cannot be handled during AUSTIN's instrumentation process. No guarantee is given by the compiler that variables will indeed be stored in a register, thus this storage specifier can be safely removed for most applications for the purpose of test data generation. Note that bit fields, where an address–of operation is also invalid, are handled as a special case by AUSTIN's source code instrumenter.

5. **Balance Branching Nodes:** This custom transformation ensures all statements of the form `if(variable)` or `if(!variable)` are explicitly turned into `if(variable != 0)` and `if(variable == 0)` respectively.

In addition to the above transformations, CIL also transforms compound predicates in the source code[1]. As a consequence, AUSTIN generates test data which satisfies the MC/DC test adequacy criterion instead of branch coverage. Figure 3.4 contains two examples showing the CIL output for code containing logical operators.

[1]One can force CIL *not* to perform this transformation, however AUSTIN uses CIL in its default mode, in which the logical 'and' and logical 'or' operators are disabled.

```
void testme(int a, int b)
{
   if(a && b)
      //target
}
```

original source code (a)

```
void foo(int a, int b)
{
   if(a)
      if(b)
         //target
}
```

CIL transformed source code(a)

```
void testme(int a, int b)
{
   int c = a && b;
}
```

original source code (b)

```
void foo(int a, int b)
{
   int c, tmp;
   if(a)
      if(b)
         tmp = 1;
      else
         tmp = 0;
   else
      tmp = 0;
   c = tmp;
}
```

CIL transformed source code (b)

Figure 3.4: Two examples illustrating how CIL transforms compound predicates into single predicate equivalents. The transformation works for both, predicates in conditional statements (see (a)) and in–line predicates (see (b)).

3.2.4 Symbolic Rewriting

In dynamic test data generation the function under test is repeatedly executed to optimize input parameters in order to reach a given goal (target node). AUSTIN instruments the source code of the unit under test to perform a pseudo–symbolic execution in parallel to the concrete execution of the unit. Since AUSTIN does not evaluate expressions symbolically it is not a classic symbolic execution. It simply re–writes operations over local variables as operations over input parameters (including globals). This often suffices for the purpose of solving pointer constraints and enables AUSTIN to use a very light–weight 'symbolic engine' that only performs symbolic assignments. Constraints over input parameters can still be collected to form a path condition describing the execution path taken by a concrete input.

Every input to the function under test (*e.g.* formal parameters, global variables and (finite) memory graphs) is modelled as a scalar symbolic variable. These variables are referred to, interchangeably, as symbolic lvalue or symbol. Symbols represent a CIL `lval` (an expression that can appear at the left of an assignment or as operand to the address–of operator) and every symbolic lvalue is mapped to a *symbolic term*. A symbolic term consists of a CIL expression, the name of the function in which it is used, and a unique identifier. Symbolic terms are unique, though multiple symbols can be mapped to the same symbolic term. CIL's `Expcompare` module can be used to check if two (CIL) expressions are equal. If they are equal and are used within the same function, then they must be represented through the same symbolic term. The equality check of expressions does not extend to their runtime value. For example, given a two dimensional array `a[2][2]`, the expressions `* (* (a + 1) + 1))` and `a[1][1]` are represented by two different symbolic terms, even though they reference the same memory location.

Figure 3.5 contains the Ocaml definitions for both a symbolic term and a symbolic lvalue. Symbolic lvalues contain a flag `isInput` which indicates whether the symbol represents an input to the function under test, a flag `global` to indicate if the symbol denotes a global variable, and `length` specifying the number of bits for symbols representing a bit field. Symbolic terms contain a flag to indicate if a CIL expression is 'uninterpreted'. For example, an expression denoting the return value of a black–box function call produces an uninterpreted symbolic term.

```
type symbolicLval = {
   lv : lval;
   funcName : string;
   isInput : bool;
   global : bool;
   length : int;
}
```

```
type symbolicTerm = {
   symbolId : int;
   funcName: string;
   mutable symbolExpr : exp;
   mutable uninterpreted : bool;
}
```

Figure 3.5: Ocaml type declarations in AUSTIN used to denote symbolic lvalues and symbolic terms. A symbolic lvalue is used to represent an input variable to the function under test, while symbolic terms provide a wrapper for CIL expressions. Symbolic terms only consist of operations involving constants or input variables to the function under test.

A symbolic store is used to maintain a state for each symbolic lvalue during the execution of the function under test. Take the example from Figure 3.1. At the start of the testing process, the symbolic store would contain the symbolic lvalue one_0 mapped to the symbolic term T_{one_0}, where the `symbolId = 1`, `symbolExpr = Lval` (one_0), `funcName = testme` and `uninterpreted = false`.

Symbolic lvalues are updated (via symbolic terms) as required during the actual execution of the function under test. An `isPointer` operator can be applied to every symbolic lvalue. It returns `true` if the symbol represents a pointer variable and `false` otherwise. The operator is an extension of CIL's built–in family of `typeOf` operators, which can be used to obtain the type of expressions or lvalues. Furthermore, every symbolic lvalue is mapped to a memory address; the address which stores the content of the lvalue represented by the symbol. Let this map be denoted by m_A. One cannot take the address of bit fields and thus AUSTIN maps symbols representing a bit field to a 'fake' memory location. The fake bit field addresses are composed of the address of the host structure containing the bit field, plus the offset of the bit field from the start of the storage of the structure. The `register` storage specifier is also removed from variables as mentioned in Section 3.2.3.

Physical memory locations change over successive executions of a function, thus the map m_A needs to be updated before and during every execution of the function under test. This update is performed during the input initialization process described in Section 3.2.7 and whenever execution enters the body of a function to allow for a correct mapping of formals and locals of a function. In addition to the address map,

AUSTIN also maintains a map m_C, that maps each symbolic term to its concrete value. This map is updated at every assignment, conditional statement, function call and return statement, as well as the start of each function entered during the execution. The latter is to initialize (update) m_C with local variables and formal parameters of the function if necessary.

As mentioned, AUSTIN does not evaluate symbolic expressions and instead uses concrete runtime values to approximate them when necessary. At every assignment statement, AUSTIN checks whether the right hand side of the assignment corresponds to one of the symbolic lvalues in its symbolic store. This is done by comparing the value of the right hand side expression with the addresses stored in m_A. When a match, s', is found, AUSTIN uses the symbolic term for s' to perform the assignment. Different symbols may be mapped to the same physical memory location in m_A due to aliasing of lvalues (expressions). When confronted with multiple matches, AUSTIN picks the first symbol found in m_A. The problem of aliases and how they are handled in AUSTIN is detailed in Section 3.2.5. When no match is found AUSTIN attempts to re–write the expression on the right hand side of the assignment in terms of input variables. This is done by providing a custom implementation of CIL's visitor interface to traverse the right hand side expression. Figure 3.6 shows a code snippet of the visitor class used for rewriting expressions. The visitor traverses an expression until it either finds a CIL lvalue which is represented by a symbolic lvalue in AUSTIN's symbolic store, encounters a constant expression, or finds an expression which cannot be re–written. An example of when AUSTIN applies its rewriting rules is shown in Figure 3.7(a), while the example in Figure 3.7(b) demonstrates a case when AUSTIN uses m_A to obtain a symbolic term for a given expression.

AUSTIN's symbolic rewriting of variables is interprocedural. Whenever the execution encounters a function call to an instrumented function, AUSTIN re–writes the arguments of the function call in terms of input parameters. Next it adds a symbolic assignment for each argument of the function call to its corresponding formal parameter in the called function. Equally if the called function contains a return value, the code is instrumented such that it performs a symbolic assignment from the return value of the function to the lvalue being assigned in the caller function. In this way operations within a called function can be expressed in terms of the input parameters of the function under test.

```
class rewriteExpClass = object(self)
   inherit nopCilVisitor

   method private handleLval (l:lval) = begin
      let termo = findSymbol l in
      match termo with
       | None -> DoChildren
       | Some(term) -> ChangeTo term.symbolExpr
   end

   method vexpr (e:exp) = begin
      match e with
       | StartOf (l) | Lval (l) -> self#handleLval l
       | SizeOfE _ | AddrOf _ | AlignOfE _ -> SkipChildren
       | _ -> DoChildren
   end
end
```

Figure 3.6: Code snippet illustrating the Ocaml class for rewriting CIL expressions in terms of input variables. The class extends CIL's default visitor, which traverses a CIL tree without modifying anything. The method `vexpr` is invoked on each occuring (CIL) expression. The subtrees are the sub–expressions, the types (for a cast or `sizeof` expression) or the variable use.

Address Map:	Symbolic Store:
$a_0 \rightarrow$ `&a`, $b_0 \rightarrow$ `&b`, $x_0 \rightarrow$ `&x`, $y_0 \rightarrow$ `&y`	$a = a_0, b = b_0, x = x_0, y = y_0$
`void testme(int a, int b) {`	
`   int x, y;`	
`   x = 2*a;`	$x_0 \leftarrow 2 * a_0$
`   y = 4*x;`	$y_0 \leftarrow 4 * (2 * a_0)$
`   ...`	
`}`	

(a) Example of when a visitor pattern can be used to re–write expressions

Address Map:	Symbolic Store:
$one_0 \rightarrow$ `&one`, $one_next_0 \rightarrow$ `&one->next`,	$one = one_0, one_next = one_next_0,$
$tmp1_0 \rightarrow$ `&tmp1`, $tmp2_0 \rightarrow$ `&tmp2`	$tmp1 = tmp1_0, tmp2 = tmp2_0$
`typedef struct item {`	
`  int v;`	
`  struct item* next;`	
`}`	
`//assume(one != (void*)0);`	
`void testme(item* one) {`	
`   unsigned long tmp1, tmp2;`	
`   tmp1 = (unsigned long )one;`	$tmp1_0 \leftarrow$ `(unsigned long)`one_0
`   tmp2 = ((struct item **)(tmp1 + 8));`	$tmp2_0 \leftarrow one_next_0$
`   ...`	because `&one->next == ((struct item **)(tmp1 + 8))`
`}`	

(b) Example of when concrete values are used to approximate symbolic expressions

Figure 3.7: The example in the top half demonstrates how symbols are updated in a symbolic store using the visitor class from Figure 3.6. The bottom row shows how AUSTIN approximates symbolic expressions with the help of concrete runtime values instead of performing symbolic pointer arithmetic. The expression `((struct item **)(tmp1 + 8))` is replaced by the expression denoting the address of `one->next`, *i.e.* `&one->next`, instead of rewriting the expression as `((struct item **)(((unsigned long)one) + 8))` using the visitor class from Figure 3.6.

If the call is to an uninstrumented function, AUSTIN marks any symbol that represents the lvalue assigned by a return statement of that function, as well as any pointer type argument to that function as uninterpreted by setting the `uninterpreted` flag. Black–box functions may change the value of their pointer type arguments.

3.2.5 Pointer Inputs

During the search process, a branch distance calculation may be required for a condition that involves a pointer comparison. However, branch distances over physical pointer addresses do not usually give rise to useful information for test data generation; for example it is difficult to infer the shape of a dynamic data structure. Therefore, instead of computing a branch distance, AUSTIN uses symbolic variables to express constraints over memory locations in a path condition, and then uses a custom solver to construct a memory graph which satisfies the constraints.

The rules that govern how the memory graph reachable from the function under test should be initialized can only be applied to constraints of the form $x = y$ and $x \neq y$, where both x or y may be the constant *null* or a scalar symbolic variable denoting a pointer input. To ensure that any code meets these requirements, AUSTIN relies on the code preparation steps described in Section 3.2.3. After the code simplification, every branching node in a control flow graph contains an expression e, consisting of a binary operation in the form of $left \bowtie right$, where $left$ and $right$ are valid CIL expressions, and $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$.

Whenever the execution traverses a branching node, AUSTIN re–writes both expressions, $left$ and $right$, in terms of input variables, applying the rules outlined in Section 3.2.4. It then adds an updated expression e' $(= left' \bowtie right')$ to the path condition. Care is taken to ensure e' describes the actual flow of execution, *i.e.* if the CFG edge denoting the false branch of the node is traversed, $\bowtie$ is inverted before adding e' to the path condition. Finally, AUSTIN updates its concrete value map m_C for $left'$ $(right')$, adding a mapping if $left'$ $(right')$ does not exist. If the symbolic term describing $left'$ $(right')$ denotes a symbolic lvalue whose address (or 'fake' address in the case of bit fields) can be taken, AUSTIN also adds a mapping from $left'$ $(right')$ to its address in m_A if it does not already exist.

A symbolic path condition *pc* is built up from aforementioned constraints in the order that they are observed during the execution of the function under test. At

this stage *pc* contains constraints over both primitive type inputs as well as pointer type inputs, and does not address the question of aliasing. When the search requires a constraint over physical memory locations to be met in order to avoid taking an infeasible path (in the CFG) with respect to the target node, AUSTIN constructs a sub–path condition, pc'. This trimmed path condition consists of constraints leading up to the critical branching node and is further simplified by removing all constraints which do not represent constraints over memory locations. For example, a constraint in which a pointer to a primitive type is dereferenced, would be removed, as would constraints over arithmetic types. Take the example in Figure 3.8 and assume node 5 is selected as target. Initially the function is executed with the input vector $\langle$`null,0`$\rangle$, producing the path condition $\langle key_0 \neq -1 \wedge one_0 = null \wedge key_0 \neq 10\rangle$. The last critical branching node in *pc* for node 5 is represented by the constraint $one_0 = null$. Thus $key_0 \neq 10$ is dropped from *pc* to form pc'. Next pc' is further reduced to only contain constraints over memory locations, leaving the single constraint $\langle one_0 = null\rangle$. Finally, the last constraint in pc' (representing the constraint of the last critical branching node) is inverted, yielding $\langle one_0 \neq null\rangle$.

From pc', AUSTIN builds an undirected equivalence graph of symbolic terms. This graph forms the basis of the pointer solving rules in AUSTIN, the idea of which has been adapted from the concolic testing tool CUTE (SMA05). The equivalence relationship between symbolic terms is defined by the '=' operators in pc', as well as aliases of symbolic terms. Figures 3.10 and 3.11 show how AUSTIN solves pointer constraints using pseudo code. Figure 3.10 deals with constructing the equivalence graph while Figure 3.11 shows how the graph is used to assign concrete values to inputs of the function under test. The remainder of this section describes the different steps in detail.

The graph is always initialized with a node denoting the symbolic term for the constant *null*, and its state is maintained across successive executions of the function under test. For each constraint *c* in pc', AUSTIN obtains the symbolic terms for the left (T_{left}) and right (T_{right}) hand sides of the binary operator in *c*. Continuing with the running example, these are $T_{left} = one_0$ and $T_{right} = null$. Next, AUSTIN collects all aliases for T_{left} and T_{right} from the maps m_A and m_C as follows. Let t denote the concrete value of T_{left} (T_{right}) in m_C, *e.g.* *null* for the running example. Find all symbolic lvalues of pointer type in m_C that are mapped to the same value

Node Id	Example
	`typedef struct item {`
	`  int key;`
	`  struct item* next;`
	`};`
	`int testme(item* one, int key)`
	`{`
(1)	`   if(key == -1)`
(2)	`      return 0;`
(3)	`   if(one != (void*)0)`
	`   {`
(4)	`      if(one->key == key)`
(5)	`         //target`
	`   }`
(6)	`   if(key == 10)`
(7)	`      ...`
	`}`

Figure 3.8: Code snippet used as a running example to explain AUSTIN's pointer solving rules.

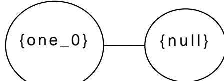
t, provided that $t \neq null$ (*i.e.* the pointer points to a valid memory location). Let this set be denoted by $s_{C_{left}}$ ($s_{C_{right}}$). Then, find all symbolic lvalues in the address map m_A that are of pointer type and whose address–of matches the value t. Let this set be denoted by $s_{A_{left}}$ ($s_{A_{right}}$). Now retrieve the symbolic term (from AUSTIN's symbolic store) for each symbolic lvalue in $s_{A_{left}}$ ($s_{A_{right}}$) to form the set $s_{T_{left}}$ ($s_{T_{right}}$). Further add the symbolic terms from $s_{C_{left}}$ ($s_{C_{right}}$) to $s_{T_{left}}$ ($s_{T_{right}}$) by forming the union $s_{T_{left}} = s_{C_{left}} \cup s_{T_{left}}$ ($s_{T_{right}} = s_{C_{right}} \cup s_{T_{right}}$). Finally check if the term T_{left} (T_{right}) is either a symbolic expression denoting a single symbolic lvalue (*e.g.* one_0) or a constant. If it is, update $s_{T_{left}}$ ($s_{T_{right}}$) such that $s_{T_{left}} = s_{T_{left}} \cup \{T_{left}\}$ ($s_{T_{right}} = s_{T_{right}} \cup \{T_{right}\}$). Otherwise leave $s_{T_{left}}$ ($s_{T_{right}}$) unchanged.

The set $s_{T_{left}}$ ($s_{T_{right}}$) now contains all aliases for T_{left} (T_{right}). Next, AUSTIN checks if a new node (containing the terms in $s_{T_{left}}$ or $s_{T_{right}}$) needs adding to the equivalence graph by checking every symbolic term in every existing node in the graph and comparing it with the elements in $s_{T_{left}}$ ($s_{T_{right}}$). If a match is found, AUSTIN updates the node with the union of the terms in $s_{T_{left}}$ ($s_{T_{right}}$) and the existing node. If no match is found, AUSTIN adds a new node with the terms from $s_{T_{left}}$ ($s_{T_{right}}$) to the graph. Let n_{left} denote the equivalence node for T_{left} and n_{right} denote the equivalence node for T_{right}. Further, let b denote the binary operator in the constraint representing the last critical branching node in pc' (*i.e.* the last constraint in pc'). If b is an inequality operator ($\neq$), and n_{left} and n_{right} are the same node, then the constraint is infeasible and AUSTIN raises an exception. Otherwise it adds an edge between n_{left} and n_{right}. Likewise, when b is an equality operator ($=$) and n_{left} and n_{right} are two different nodes, AUSTIN also raises an infeasible constraint exception. The evolution of the graph with a more elaborate example is shown in Figure 3.9.

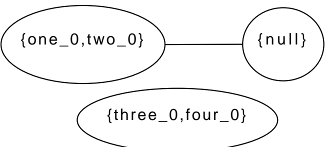
Note, before adding a new node to the equivalence graph, AUSTIN checks that the node does not denote an empty set. This could occur when a constraint still contains pointer arithmetic that cannot be replaced by a symbolic lvalue because the resulting memory location is unavailable to AUSTIN. When this happens, AUSTIN forces a random restart, removing all existing nodes from the equivalence graph, with the hope of traversing a different path, leading to a new set of more amenable constraints.

Once the equivalence graph has been built, AUSTIN needs to assign concrete values to the lvalues denoted by the symbolic terms in the graph. The rules for doing this are

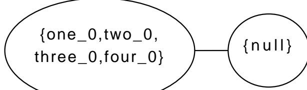
Node Id	Example
	`void testme(item* one, item* two, item* three, item* four) {`
(1)	`if(one != (void*)0)`
(2)	`if(one == two)`
(3)	`if(three == four)`
(4)	`if(one == three)`
(5)	`//target`
	`}`



Equivalence graph used to generate inputs for the path which executes the decision node 1 as true



Equivalence graph used to generate inputs for the path which executes the decision nodes 1, 2, 3 as true



Equivalence graph used to generate inputs for the path which executes all decision nodes as true

Figure 3.9: An example illustrating how AUSTIN builds up an equivalence graph of symbolic terms over successive executions. The graph is used to instantiate the concrete pointer type inputs to the function under test. The example shows the evolution of the graph when node 5 has been selected as target.

implemented in the procedure `apply_rules` which is shown in Figure 3.11. The nodes n_{left} and n_{right} as well the binary operator b form the inputs to that procedure.

If b denotes the arithmetic comparison operator '$\neq$', then n_{left} and n_{right} must be two separate nodes. Further, at most one node may contain a constant, otherwise AUSTIN cannot solve the constraint and forces the search to perform a random restart[1]. Assuming this property holds, AUSTIN selects a symbolic term at random from the node which does not contain a constant. If neither node contains a constant, a symbolic term from n_{left} is chosen. For the running example this would be the term one_0. A new memory location (via `malloc`) is then assigned to the lvalue denoted by the chosen term. All lvalues represented by symbolic terms in the same node are also pointed to that location. Finally the equivalence graph needs updating by adding an edge between the node containing the updated terms, and the node containing the symbolic term *null* if it does not already exist.

If b denotes the arithmetic comparison operator '$=$', then n_{left} and n_{right} must be the same node and AUSTIN proceeds as follows. If the node n_{left} contains a constant, all lvalues denoted by the symbolic terms in n_{left} are assigned that constant. This applies to compile time constants as well as runtime constants (*e.g.* expressions taking the address of a global variable). Else if n_{left} contains an edge to the node containing *null*, AUSTIN chooses a symbolic term from n_{left} and assigns a new memory location to the lvalue denoted by that term. All lvalues represented by the terms in n_{left} are assigned the same location. Otherwise AUSTIN assigns the constant *null* to all lvalues denoted by the symbolic terms in n_{left}.

Finally, AUSTIN iterates over all remaining nodes in the graph that contain an edge to the node denoting the constant *null*. From each of these nodes it chooses one symbolic term T and assigns `malloc` to the lvalue denoted by T. Then all lvalues denoted by the remaining terms in that node are pointed to the same memory address.

3.2.5.1 Limitations of Pointer Rules

The proposed approach has a few limitations which are set out in the remainder of this section. Consider the code fragment shown below:

[1] Note the case where $constant1 \neq constant2$ holds does not apply because it would not invoke a constraint solving move in AUSTIN.

Global inputs: $force_random_restart$; $EG = \langle N, E \rangle$: equivalence graph of symbolic terms; N: nodes; $E \subseteq N \times N$: edges; m_C: map of symbolic terms to their concrete value; m_A: map of symbolic lvalues to their address

solve_constraint (s : *candidate_solution*)

pc := path condition describing execution path taken by s

pc' := sub–path condition extracted from pc; remove all constraints that appear after the last occurrence of a branching node and remove all constraints which do not specify a constraint over memory locations

invert last constraint in pc'

add a node containing the symbolic term for *null* to *EG* if it does not exist

for each constraint c **in** pc' **do** //c is of the form $T_{left} \bowtie T_{right}$

$s_{C_{left}}$:= collect all concrete aliases of T_{left} from m_C

$s_{C_{right}}$:= collect all concrete aliases of T_{right} from m_C

$s_{A_{left}}$:= collect all concrete aliases of T_{left} from m_A

$s_{A_{right}}$:= collect all concrete aliases of T_{right} from m_A

$s_{T_{left}}$:= union of set of symbolic terms for the symbolic lvalues in $S_{A_{left}}$ and $s_{C_{left}}$

$s_{T_{right}}$:= union of set of symbolic terms for the symbolic lvalues in $S_{A_{right}}$ and $s_{C_{right}}$

if isSymbolicLvalue(T_{left}) **or** isConstant(T_{left}) **then**

$s_{T_{left}} := s_{T_{left}} \cup T_{left}$

end if

if isSymbolicLvalue(T_{right}) **or** isConstant(T_{right}) **then**

$s_{T_{right}} := s_{T_{right}} \cup T_{right}$

end if

n' := find or create (and add) *EG* node that contains one of the symbolic terms in $s_{T_{left}}$

$n_{left} := n' \cup s_{T_{left}}$

n' := find or create (and add) *EG* node that contains one of the symbolic terms in $s_{T_{right}}$

$n_{right} := n' \cup s_{T_{right}}$

if ($n_{left} = \emptyset$ **and** $n_{right} = \emptyset$) **or**

($n_{left} \neq n_{right}$ **and** containsConstant(n_{left}) **and** containsConstant(n_{right})) **then**

$force_random_restart := true$; return s

else if ($n_{left} = n_{right}$ **and** op = '$\neq$') **or**

($n_{left} \neq n_{right}$ **and** op = '$=$') **then**

raise exception "Infeasible constraint"

else if op = '$=$' **then**

update *EG* with $n_{left} \cup n_{right}$

else if op = '$\neq$' **then**

update n_{left} and n_{right} in *EG*

add edge between n_{left} and n_{right}

end if

if $c = p'$ **then**

s' := apply_rules($n_{left}, n_{right}, \bowtie$)

perform *evaluation*

if requires_constraint_move(s') **and** $evaluations < max_evaluations$ **then**

s' := solve_constraint(s')

end if

return s'

end if

end for each

Figure 3.10: Pseudo code illustrating the algorithm for constructing an equivalence graph. The graph forms the basis for deriving values assigned to pointer inputs of a function under test.

```
Global inputs: EG =⟨N, E⟩: undirected equivalence graph of symbolic terms; N : nodes;
E ⊆ N × N : edges
apply_rules (n_left : set of symbolic terms, n_right : set of symbolic terms, op :
binary operator)
 if op = "≠" then
   if containsConstant(n_left) then
     item := choose element from n_right
     value := malloc(sizeof(item))
     assign value to all lvalues denoted by elements in n_right
     add edge in EG from EG node n_right to node containing null
   else
     item := choose element from S_left
     value := malloc(sizeof(item))
     assign value to all lvalues denoted by elements in S_left
     add edge in EG from EG node n_left to node containing null
   end if
 else if op = "=" then
   S_U := S_left ∪ S_right
   if containsConstant(S_U) then
     item := choose constant from S_U
     assign item to all lvalues denoted by elements in S_U
   else
     n := node ∈ N containing terms in S_U
     if hasEdgeToNull(n) then
       item := choose element from S_U
       value := malloc(sizeof(item))
       assign value to all lvalues denoted by elements in S_U
     else
       assign null to all lvalues denoted by elements in S_U
     end if
 end if
 for each EG node n ∈ N do
   if not(containsTerms(n, S_left)) && not(containsTerms(n, S_right)) then
     if hasEdgeToNull(n) then
       item := choose symbolic term from n
       value := malloc(sizeof(item))
       assign value to all lvalues denoted by symbolic terms in n
     end if
   end if
 end for each
```

Figure 3.11: Pseudo code illustrating the algorithm for assigning concrete values to pointers. The manner in which new or existing memory locations are assigned to a function's pointer inputs depends on the structure of the equivalence graph.

```
void foo(int* ip1, int* ip2) {

   if(ip2 - ip1 == 3)
      //target
}
```

AUSTIN is not able to generate inputs to the function that would satisfy the condition of the `if` statement, *i.e.* allocate an array of at least 3 elements, then point `ip1` and `ip2` somewhere within the array, 3 elements apart (or `ip2` just beyond the end of the array).

The second limitation occurs in the presence of loops. Consider the two slightly different code fragments shown below:

```
typedef struct item {
   int val;
   struct item* next;
};
void foo(item* p) {

   while(p != null) {

     if(p->val == 2) {
        //target
     }
     p = p->next;
   }
}
```

```
typedef struct item {
   int val;
   struct item* next;
};
void foo(item* p) {
   int counter = 0;
   while(p != null) {
     counter = counter + 1;
     if(counter == 5) {
        //target
     }
     p = p->next;
   }
}
```

While AUSTIN can generate an input to traverse the true branch of the `if` statement in the left column within 5 fitness evaluations, it fails to find an input to do the same for the code in the right column. A previous version of AUSTIN allowed loops to be unrolled infinitely many times. Thus, with every execution of the function, AUSTIN increased the length of the list denoted by the input parameter `p` by one item. As a consequence, the number of primitive type inputs also grew by one in every iteration, leading to an increase in the size of the input domain for the AVM. An increased input domain makes it more difficult for a search to find a solution, although for the particular example shown in the right column, AUSTIN was able to find a desired input (*i.e.* a list of length 5) 'by chance'. The new version of AUSTIN no longer 'blindly' unrolls

loops and thus does not unnecessarily increase the size of the input domain. Instead it generates an input which enters the body of the loop just once. Then it gets stuck trying different values for the input parameter `p->val` until its fitness budget has been exhausted. This behaviour will be corrected in future versions of AUSTIN. Before attempting a random restart, AUSTIN will include a check if a loop condition is a critical branching node for the current target. If it is, AUSTIN will attempt to increase the number of loop iterations, *e.g.* by extending the list in the above example. If this does not result in an improved fitness value, AUSTIN will continue with a random restart. Otherwise it will continue with its loop unfolding.

Another limitation is that every constraint needs to be specified in the code in the form of `if` statements. For example, AUSTIN cannot infer preconditions to functions. One common precondition observed while evaluating AUSTIN is that pointers must point to a valid memory location in order to prevent segmentation faults, and thus a premature conclusion of the test data generation process. This will be discussed in more detail in Chapter 4.

3.2.6 The Algorithm

The top level algorithm implemented in AUSTIN is shown as pseudo code in Figure 3.12. Search algorithms commonly require a user to set an upper bound of fitness evaluations (*i.e.* how many times the function under test may be executed) in case the search fails to find a solution. The default fitness allowance used in AUSTIN is $10,000$ evaluations per branch. This limit can be adjusted by a user. The phrase '*perform evaluation*', used throughout the pseudo code describing the different algorithms implemented by AUSTIN, groups together the following actions: execution of a candidate solution, recording of the execution trace produced by the solution and incrementing the evaluation count.

The method *obj* used as part of the pseudo code in Figures 3.12, 3.14 and 3.15 denotes the objective function used in AUSTIN. It returns a value, based on *branch distance* and *approach level*, two common metrics used in search–based testing. The objective (or fitness) value is used to determine if a neighbour is better than the current individual. The remainder of this section summarizes the various steps of AUSTIN's algorithm.

```
Global Inputs: evaluations, max_evaluations, input_index, force_random_restart,
increased_precision, precision_success
Initialize: evaluations := 0, input_index := 0, force_random_restart := false,
increased_precision := false, precision_success := false
hill climb
  s := random
  perform evaluation
  while obj(s) > 0 and evaluations < max_evaluations do
    if requires_constraint_move(s) and not(force_random_restart) then
      s' := solve_constraint(s)
      if obj(s') < obj(s) then
        s := s'
      else
        force_random_restart := true
      end if
      reset_exploration_parameters()
    else if neighbourhood_explored(s) or force_random_restart then
      s := random
      reset_exploration_parameters()
      force_random_restart := false
      perform evaluation
    else
      primitives := vector of all primitive type variables in s
      v := primitives.at(input_index)
      if typeOf(v) = enum type then
        s := make_enum_move(s)
      else
        s' := explore(s)
        if obj(s') < obj(s) then
          s := s'
          if obj(s) > 0 and evaluations < max_evaluations and
              not(requires_constraint_move(s)) then
            if increased_precision then
              precision_success := true
            end if
            repeat_iteration := 1
            s := pattern_move(s, repeat_iteration)
          end if
        end if
      end if
  end while
```

Figure 3.12: Pseudo code showing the top level search algorithm implemented in AUSTIN. It extends a hill climber to include constraint solving rules for pointer inputs to a function under test. The steps for handling numerical type inputs to a function under test are based on the alternating variable method introduced by Korel (Kor90).

First a candidate solution is generated by assigning zero to all primitive type input parameters and *null* to all pointer inputs of a function under test. AUSTIN keeps a separate vector of all primitive type variables (*e.g.* `int, float` *etc.*) in a candidate solution. During the test data generation process, a series of exploratory and pattern moves, described in Section 2.1.2, is performed on this vector. The size of the vector only changes when AUSTIN changes the size of the input domain for the function under test, *e.g.* by assigning *null* or a non–*null* value to a pointer variable. The pseudo code of the methods implementing the exploratory and pattern moves in AUSTIN can be found in Figures 3.13 and 3.14 respectively.

Experiments have shown that exploratory moves are ineffective for n–valued enumeration type variables, variables that can only take on n discrete values. AUSTIN therefore exhaustively explores all n values of an enumeration type. The pseudo code is shown in Figure 3.15. When a particular value from an enumeration list leads to a better neighbour, AUSTIN adapts it as the current individual and continues with exploratory moves for the other elements in its input vector. This approach seems to scale well in practice.

After every concrete execution of a candidate solution AUSTIN checks if it needs to apply its pointer solving procedure described in Section 3.2.5, or the hill climb search laid out in Section 2.1.2 in order to move closer to the target node. The decision (shown as pseudo code in Figure 3.16) depends on the constraints contained within the last critical branching node where execution diverged away from the target. The `targetOf` operation in Figure 3.16 returns the symbolic term which represents the target of a pointer. The function `isPointerDereference` decomposes a symbolic term into its CIL lvalue (if any), and checks if that lvalue denotes a pointer dereference. By default this procedure returns `true`, *e.g.* when a symbolic term does not contain a CIL lvalue.

Finally, the procedure `neighbourhood_explored` shown as pseudo code in Figure 3.2 is used to determine if AUSTIN is stuck on a local optimum and requires a random restart. If no change in value of any of the primitive type variables leads to an improvement in fitness, and no constraints over physical memory location remain to be solved, AUSTIN will seek to generate a new random solution[1], thereby sampling different regions of the search–space. However, before restarting, AUSTIN checks if the best candidate solution found thus far contains any floating point variables. If it does,

[1]Pointers are always initialized to *null*.

```
Global Inputs: input_index, direction, last_direction, last_input_index
explore (s : candidate_solution)
  last_direction := direction
  last_input_index := input_index
  primitives := vector of all primitive type variables in s
  v := primitives.at(input_index)
  v := v + (direction * 10^(-acc(v)))
  update v in primitives
  s' := (s with updated primitives)
  if direction < 0 then
    direction := 1
  else
    direction := −1
    increment input_index
  end if
  perform evaluation
  return s'
```

Figure 3.13: Pseudo code illustrating the algorithm for performing exploratory moves which are part of the alternating variable method.

Global Inputs: *evaluations*, *max_evaluations*, *direction*, *last_direction*, *input_index*, *last_input_index*

pattern_move (s : *candidate_solution*, *iteration* : *int*)
  primitives := vector of all primitive type variables in s
  $v := primitives.at(last_input_index)$
  $v := v + (last_direction * 10^{-acc(v)} * 2^{iteration})$
  update v in *primitives*
  $s' :=$ (s with updated *primitives*)
  perform *evaluation*
  if $obj(s') < obj(s)$ **then**
    $s := s'$
    if requires_constraint_move(s) **then**
      return s
    else if $obj(s) > 0$ **and** $evaluations < max_evaluations$ **then**
      $s :=$ pattern_move(s, $iteration + 1$)
    end if
  end if
  reset_exploration_parameters()
  return s

Figure 3.14: Pseudo code illustrating the algorithm for performing pattern moves in the alternating variable method. Pattern moves are designed to speed up the search by accelerating towards an optimum.

```
Global Inputs: input_index, evaluations, max_evaluations
make_enum_move (s : candidate_solution)
  primitives := vector of all primitive type variables in s
  v := primitives.at(input_index)
  items := get number of items in enum list for v
  increment input_index
  for i := 0 to i < items and evaluations < max_evaluations do
    set v to the i^th item in enum list for v
    update v in primitives
    s' := (s with updated primitives)
    perform evaluation
    if obj(s') < obj(s) then
      reset_exploration_parameters()
      return s'
    end if
    increment i
  end for
  return s
```

Figure 3.15: Pseudo code illustrating the algorithm for optimising n–valued enumeration type variables, variables that can only take on n discrete values. Whenever an enumeration type variable is to be optimized, the algorithm iterates over all n values (in the order they have been declared in the source code).

```
Global Inputs: CFG =⟨N, E⟩: control flow graph of function F; N: nodes; E ⊆
N × N: edges; target ∈ N: current target node; PC: path condition
requires_constraint_move (s : candidate_solution)
  t := execution trace produced by s
  if target appears in t then
    return false
  end if
  nid := CFG node id which is the last critical branching node in t with respect to
target
  p := predicate associated with nid in PC of the form: T_left ⋈ T_right
  if ⋈ ∈ {=, ≠} then
    if ( isPointerDereference(T_left) and
          isPointer(targetOf(T_left)) = false ) or
        ( isPointerDereference(T_right) and
          isPointer(targetOf(T_right)) = false ) then
      return false
    else if isPointer(T_left) = true and
              isPointer(T_right) = true then
      return true
    else
      return false
    end if
  else
    return false
  end if
```

Figure 3.16: Pseudo code illustrating the method for checking if AUSTIN should use its custom constraint solving rules for pointers, or the alternating variable method to satisfy the constraint in a branching node.

Global Inputs: $EG = \langle N, E\rangle$: undirected equivalence graph of symbolic terms; N : nodes; $E \subseteq N \times N$: edges; *input_index*, *direction*
reset_exploration_parameters ()
  $direction := -1$
  $input_index := 0$
  remove all nodes from EG

Figure 3.17: Pseudo code illustrating the reset operation performed by AUSTIN at every random restart.

AUSTIN will try and increase the accuracy with which floats are optimized to see if an increased accuracy is sufficient to escape from the local optimum. Whenever the accuracy of a float is increased, AUSTIN restarts its exploratory moves for the current individual. If a change in accuracy has no positive effect on the fitness value of an individual, the change is reverted, and AUSTIN tries to increase the accuracy of the next floating point variable. Once the accuracy of each floating point input has been increased but without yielding a better neighbour, AUSTIN proceeds with the random restart. After every restart, AUSTIN uses the procedure `reset_exploration_parameters` shown in Figure 3.17 to reset its search parameters.

3.2.7 Input Initialization

Inputs to the function under test are initialized recursively from within the instrumented source code. AUSTIN examines global variables in the source file and all formal parameters of the function under test. Based on the two lists of variables, it automatically constructs a test driver which serves as an entry point to the unit under test. All inputs (globals and formals) are initialized through that function. Figure 3.18 shows the pseudo code for the top–level input initialization used in AUSTIN. Its input is a candidate solution consisting of all global and formal parameters for the function under test, as well as variables denoting different regions of the input memory graph. The CIL function `unrollType` returns the type of a variable, resolving named user defined types. For example, the type of `MYCHAR` in the following declaration `typedef char MYCHAR` would be resolved to `char`. The `addrOrStartOf` function is a wrapper for CIL's `mkAddrOrStartOf`, which constructs an expression denoting the

```
initialize_solution(s : candidate_solution)
  foreach v in s do
    t := unrollType(typeOf(v))
    if t = data structure or union then
      generate_struct_or_union_input(t, addrOrStartOf(v))
    else
      v := initialize_input(t, v)
    end if
    update_input_map(v)
  end foreach
```

Figure 3.18: Pseudo code illustrating the entry point for initializing a candidate solution (*i.e.* the inputs to the function under test) with concrete values.

start of the storage denoted by a variable. The wrapper implements the 'fake address' handling for bit fields.

Figure 3.19 shows the procedure `initialize_input` which uses the type of a variable to generate a concrete value for it. First it checks if the variable has already been initialized. If it has, its current value is returned by one of the sub–procedures shown in Figure 3.20. Otherwise the variable is initialized according to AUSTIN's current configuration: for example, a user may specify that primitive inputs should be initialized with random values. Structure (and union) types are initialized by passing each field of the structure (or union) to `initialize_input`, or the function `generate_struct_or_union_input` in case of nested structures and union types. Finally, every time an input is assigned a concrete value, the concrete map m_C and address map m_A are updated as shown in Figure 3.21.

3.2.7.1 Limitations of Input Types

AUSTIN currently has some limitations when generating input values for the function under test. Firstly, void pointers are always non–randomly assigned *null*, and AUSTIN has no way of changing their value. Secondly, function pointers are also assigned the constant *null*. AUSTIN does not attempt to assign the address of functions from the source code that would form potential matches. Users may edit the test driver generated by AUSTIN to provide their own implementation of how to instantiate such

```
initialize_input(t : data type, v : variable)
  if not(initialized(v)) then
    initialize := true
  else
    initialize := false
  end if
  if t = enum type then
    return generate_enum_input(v, t, initialize)
  else if t = primitive type then
    return generate_primitive_input(v, t, initialize)
  else if t = pointer type then
    v := generate_pointer_input(v, initialize)
    update_input_map(v)
    if v ≠ null then
      if unrollType(typeOf(targetOf(v))) = data structure or union then
        generate_struct_or_union_input(v)
      else
        ∗v := initialize_input(unrollType(typeOf(∗v)), ∗v)
        update_input_map(∗v)
      end if
    end if
    return v
  else
    failwith "unrecognised input type"
  end if
```

Figure 3.19: Pseudo code illustrating the method assigning concrete values to an input variable of the function under test.

```
Global Inputs: init_to_zero
generate_primitive_input(v : variable, t : data type, initialize : boolean)
  if initialize = true then
    if init_to_zero = true then
      return 0
    else
      return random_number(min(t), max(t))
    end if
  else
    return v
  end if

generate_enum_input(v : variable, t : data type, initialize : boolean)
  if initialize = true then
    items := list of items with names and values from t. This list
              should be non-empty. The item values must be compile–time constants
    return items.at(0)
  else
    return v
  end if

generate_struct_or_union_input(v : pointer to variable)
  foreach member m in *v do
    t := unrollType(typeOf(v → m)
    if t = data structure or union then
      generate_struct_or_union_input(t, addrOrStartOf(v → m)))
    else
      v → m := initialize_input(t, v → m))
    end if
    update_input_map(*v)
  end foreach

generate_pointer_input(v : pointer variable, initialize : boolean)
  if initialize = true then
    return null
  else
    return v
  end if
```

Figure 3.20: This group of methods shows the pseudo code for assigning values to inputs of the function under test, based on their type declaration.

Global Inputs: m_C: map from symbolic terms to their concrete values; m_A: map from symbolic lvalues to the start of the storage denoted by the symbolic lvalue ('fake' address for bit fields)
update_input_map(v : *variable*)
  sl := find or create symbolic lvalue for v
  st := find or create symbolic term from sl
  mark v as initialized
  add mapping from sl to addrOrStartOf(v) to m_A
  add mapping from st to v to m_C

Figure 3.21: Pseudo code illustrating how AUSTIN instantiates its map of concrete values m_C and address map m_A at the start of every execution of the function under test. The maps are also updated during the dynamic execution of the function under test.

inputs. Another shortcoming of AUSTIN occurs in the presence of `union` constructs. Currently, AUSTIN can only generate input values for the last declaration appearing in a union because it does not differentiate between structures and unions (see the procedure `generate_struct_or_union_input` in Figure 3.20). In order to generate meaningful values for a union type AUSTIN would require an analysis to determine which member of a union is being used for a given execution path. This analysis has not yet been implemented.

A further limitation concerns arrays (such as strings), which are denoted by a pointer, because AUSTIN can only handle fixed–length arrays. Thus for functions which contain an array as formal parameter, a tester may need to change the functions declaration from a pointer notation to an array notation in order for AUSTIN to know the size of the array. Furthermore, in the case of strings, a tester also has to edit the test driver in order to manually append a terminating null character to a string.

3.2.8 Usage

AUSTIN has been implemented as a command line tool for Linux and is available at `http://www.dcs.kcl.ac.uk/pg/kiran/software/index.php`. One can specify a target branch, fitness budget and how to initialize primitive type inputs. AUSTIN can also merge multiple source files into a single file using CIL's `Mergecil` module. AUSTIN is run in two stages: first, a (merged) source file is instrumented and information about

the function under test is saved in binary data files. Then the instrumented file needs to be compiled by a user into a shared library, after which AUSTIN is ready to start the test data generation process.

AUSTIN loads the unit under test, *e.g.* a set of functions or even a complete application, as a shared library. As a result both AUSTIN and the unit under test run in the same address space. This can often lead to problems, *e.g.* when the function under test crashes due to an exception such as a segmentation fault. Future versions of AUSTIN will use separate processes for AUSTIN and the unit under test. This will avoid AUSTIN being affected by exceptions raised while executing the unit under test. Furthermore, it enables AUSTIN to use certain exceptions, *e.g.* segmentation faults, to guide its input generation.

3.3 Evolutionary Testing Framework

The evolutionary testing framework was developed as part of the EvoTest project (GKWV09) (IST–33472), applying evolutionary algorithms to the problem of testing software systems. The ETF supports both black–box and white–box testing and represents the state–of–the–art for automated evolutionary structural testing. The framework is specifically targeted for use within industry, with much effort spent on scalability, usability and interface design. It is provided as an Eclipse plugin, and its white–box testing component is capable of generating test cases for single ANSI C functions. A full description of the system is beyond the scope of this thesis and the interested reader is directed towards the EvoTest web page located at `www.evotest.eu`.

At its core, the ETF contains a user configurable evolutionary engine, which has been integrated from the GUIDE (CS09) project. The framework also implements a subset of the approach introduced by Prutkina and Windisch (PW08) to handle pointers and data structures. It maintains different pools of variables, which are used as the target of pointers and whose value is optimized by the evolutionary search. Each pool contains the subset of global variables and formal parameters of the function under test that are of the same data type. In addition, for parameters denoting a pointer to a primitive type or data structure, the ETF creates a temporary variable whose type matches the target type of the pointer. These temporary variables are also added to the pools.

Each variable in the pools is assigned an index in the range $0, ..., n-1$, where n is the size of their pool (*i.e.* the number of variables in the pool). The individual (chromosome) describing a pointer input to the function under test contains two fields; one denoting an index and the other a value. The index is used to select a variable from the (correct) pool whose address is used as the target for the pointer input. Note that an index may be negative to denote the constant *null*. Take the code snippet from Figure 3.22, and assume that the part of the chromosome describing the formal inputs to `testme` is of the form $\langle(0,5),(1,6)\rangle$, forming a (index, value) pair for each pointer input[1]. To test this function the ETF only requires one pool of variables, those of type `int`. The pool contains the global variable `x` (at index 0) and two temporary variables, `var_p` (at index 1) and `var_q` (at index 2). The above chromosome is interpreted by the ETF as follows. The index of the first pair (0) tells the ETF to use the address of the variable `x` as target for the formal parameter `p`. The value part of the pair is assigned to the temporary variable `var_p`. The index of the second pair (1) informs the ETF to use the address of the variable `var_p` as target for the pointer `q`. Its value part is assigned to the temporary variable `val_q`, which remains unused in this example. In this way the ETF is able to generate pointers initialized to *null*, the same value and different values. An important difference to AUSTIN is the fact that the ETF cannot handle pointers to recursive types such as lists and graphs.

3.4 Empirical Study

The empirical study consisted of 8 C functions, selected from three case studies. The case studies had been selected by Berner & Mattner Systemtechnik GmbH to form part of the evaluation of the ETF within the EvoTest (GKWV09) project. Each case study consists of an embedded software module from the automotive area generated using one of two popular code–generation tools as described in Table 3.1. The eight functions had been selected to provide a representative sample of their respective case studies, with particular attention paid to the number of branches and nesting level. Table 3.2 gives a breakdown of relevant metrics for the selected functions. These functions formed a benchmark to compare AUSTIN against.

[1] Chromosomes are passed in XML format to and from the evolutionary algorithm.

<table>
<tr><td colspan="2">Example Function:</td></tr>
<tr><td colspan="2"><code>int x;</code>
<code>void testme(int* p, int* q) {</code>
<code>...</code>
<code>}</code></td></tr>
<tr><td>Concrete input instantiation:
<code>int val_p = 5;</code>
<code>int val_q = 6;</code>
<code>p = &x;</code>
<code>q = &val_p;</code></td><td>Example chromosome:
$(0, 5), (1, 6)$

<code>//x is at index 0</code>
<code>//val_p is at index 1</code></td></tr>
</table>

Figure 3.22: Example showing how the ETF initializes pointers to primitive types. The chromosome only shows the parts which relate to the two pointer inputs `p` and `q`. The first component of each pair denotes an index (of a variable in a pool) while the second component denotes a value.

Table 3.1: Case studies. LOC refers to the total preprocessed lines of C source code contained within the case studies. The LOC have been calculated using the CCCC tool (Lit01) in its default setting. Table 3.2 shows the individual per–function LOC metric.

Case Study	LOC	Functions Tested	Code Generation Tool	Software Module
B	18,200	02, 03, 06	dSpace TargetLink (dT)	Adaptive headlight control
C	7,449	07, 08, 11	ETAS ASCET (ASC)	Door lock control
D	8,811	12, 15	ETAS ASCET	Electric window control
Total	34,460			

In order to extend the applicability of the study, a number of publicly available dynamic symbolic execution based testing tools for C, CREST (BS08) and CUTE (SMA05) were investigated. The goal was to establish whether they could be applied to test subjects in Table 3.1. Unfortunately neither CREST nor CUTE could be applied because their instrumentation produced uncompilable code.

All functions from the case studies make extensive use of bit fields within data structures. Bit fields can only be declared inside a data structure or union, and can be used to specify small objects of a given number of bits in length. They do not have addresses and you cannot have pointers to them or arrays of them, thus one cannot apply the address–of operator (`&`) to a bit field.

Both CREST and CUTE instrument the source code to take the address of variables, including the addresses of members of a data structure. CREST uses the addresses of variables as identifiers, maintaining a map from addresses to symbolic expressions. The precise use of an address of a variable in CUTE is not known. Nevertheless, both CUTE and CREST cannot be applied to source code containing the use of bit fields. CUTE is no longer maintained, however this issue has been reported to the developers of CREST via the Google code website (issue ID 2). In response, the developers of CREST stated that this bug may be fixed in a future version, but that it remains unresolved for the time being. However, as a consequence of the bug report they have updated CREST's documentation to note the problem with bit fields.

Having restricted the scope of the empirical study to AUSTIN and the ETF, the purpose of the study was twofold. The first aim was to check whether AUSTIN could be applied 'out of the box' to real industrial code. AUSTIN requires a tester to manually compile the source code under test into a shared library. Employees at Berner & Mattner Systemtechnik GmbH raised the concern that not everybody may be familiar with compiling source code from the command line. In contrast, the ETF contains a cross–platform Eclipse–based ETF structural test component which is much more user–friendly and applicable to industrial use. It should be acknowledged that, in contrast to AUSTIN, whose primary motivation was to provide a research prototype, industrial applicability was a prime driver of the ETF project. One area in which both tools could be improved, is the output of the test data. Neither tool currently outputs the generated test cases in a manner that can easily be integrated into an existing testing framework.

Table 3.2: Test subjects. The LOC have been calculated using the CCCC tool (Lit01) in its default setting. The number of input variables counts the number of independent input variables to the function, *i.e.* the member variables of data structures are all counted individually.

Obfuscated Function Name	LOC	Branches	Nesting Level	Nr. Inputs	Has Pointer Input
02	919	420	14	80	no
03	259	142	12	38	no
06	58	36	6	14	no
07	85	110	11	27	yes
08	99	76	7	29	yes
11	199	129	4	15	yes
12	67	32	9	3	no
15	272	216	4	28	yes
Total	1,958	1,161			

Once AUSTIN was setup, the second objective of the study was to investigate the effectiveness and efficiency of AUSTIN when compared to the ETF.

Effectiveness of AUSTIN

Previous work has compared the AVM with a GA in the context of branch coverage testing (HM09). However, this is the first study to compare two search–based techniques that use different ways of automatically handling pointer inputs. To the best of the author's knowledge this is also the first study which compares two search–based testing tools on machine generated code. Thus, based on previous work by Harman and McMinn (HM09) which had found the AVM to be more effective than an evolutionary search, the goal was to verify their findings which led to the following null and alternate hypotheses:

H_0 : *AUSTIN is as effective as the ETF in achieving branch coverage.*
H_A : *AUSTIN is more effective than the ETF in achieving branch coverage.*

Efficiency of AUSTIN

Alongside coverage, efficiency is also of paramount importance especially in an indus-

trial setting. As with coverage, Harman and McMinn (HM09) found an AVM to be statistically significantly more efficient than a GA for branches which could be covered by either technique. Often one does not know *a priori* how successful one technique is going to be in achieving coverage. For example if a (search–based) technique is ineffective it will also be inefficient because it will run until its fitness budget has been exhausted. The goal therefore was to compare the efficiency of AUSTIN and the ETF in an industrial setting, which led to the formulation of the following null and alternative hypotheses:

> H_0 : *AUSTIN is equally as efficient as the ETF in achieving branch coverage of a function.*
> H_A : *AUSTIN is more efficient than the ETF in achieving branch coverage of a function.*

3.4.1 Experimental Setup

The data for the experiments with the ETF on the functions listed in Table 3.2 had already been collected for the evaluation phase of the EvoTest project (GKWV09). This section serves to describe how the ETF had been configured and how AUSTIN was adapted to ensure as fair a comparison as possible between AUSTIN and the ETF.

Every branch in the function under test was treated as a goal for both the ETF and AUSTIN. The order in which branches are attempted differs between the two tools. AUSTIN attempts to cover branches in the function under test in the reverse breadth first order they appear in the CFG, while the ETF attempts branches in a depth first order from the start of the function under test. In both tools branches that are covered serendipitously while attempting a goal are removed from the list of goals. The fitness budget, *i.e.* the maximum number of times the function under test may be executed, was set to $10,000$ evaluations per branch.

The ETF contains a set of default parameters for its evolutionary engine. A user may edit this configuration to adapt the evolutionary engine to a particular problem. For the purpose of this study, the ETF was configured to use a GA whose parameters were manually tuned to provide a good set which was used for all eight functions. The GA was set up to use a population size of 200, deploy strong elitism as its selection strategy, use a mutation rate of 1%, and a crossover rate of 100%.

In addition to allowing a user to configure the parameters of the evolutionary search, the ETF also provides the option to reduce the input domain size by narrowing the bounds of the input variables, or completely excluding them from the search. The resulting smaller input domain leads to quicker searches, assuming that the bounds have been set such that all branches can still be feasibly covered. Bounds reduction can be exploited to harness testers' knowledge of the semantic properties of input variables – for example a particular input variable may represent the state of a finite state machine, in which case for example only the values 0 to 7 of an unsigned 16–bit integer are necessary to cover all branches in the function. By setting appropriate bounds for this single variable alone, the input domain size can be reduced by a factor of $8,192$.

Despite the fact that little semantic knowledge of the input variables was available (since the functions came from operational systems unfamiliar to the tester), it was possible to use bounds reduction to substantially reduce the size of the input domain as detailed in the following paragraphs.

Having selected a function to generate test cases for, the ETF performs a pseudo variable dependence analysis for the function with respect to global variables. More precisely, it conducts a syntactic check on global variables appearing in predicates. If a global variable does not appear in any predicate it is removed from the list of inputs. This dependence analysis does not include a dataflow or alias analysis, and thus is unreliable. However, this behaviour of the ETF is built in and unchangeable. It is worth mentioning that the developers of the ETF are working on a correct variable dependence analysis for global variables.

After the automatic variable elimination, the ETF displays the remaining input variables for the function in a bounds reduction window. The following heuristics were used to reduce the input domain of the remaining variables even further.

Elimination of input variables: Since the automatic parameter reduction performed by the ETF operates only at the whole variable level there were many situations where a complete structure is included in the search by default, even though only a minority of its members are relevant for branching decisions. By manual inspection of the code, where practical, it was possible to determine which member

variables were not relevant to the control flow. Variables that the tester determined with certainty were not relevant for branching decisions (by means of this manual data flow analysis), were removed from the input domain.

Bounds reduction of 'boolean' variables: Since C does not contain a boolean type [1], such variables are often implemented as integers. Through inspection of the code it is possible to identify these variables and set their bounds to $[0, 1]$. The following indicators for boolean–type integers were used:

- `typedef` definitions such as `typedef int Bool`
- variable name prefixes indicating that the variable is used as a boolean, *e.g.* `short blVariable`
- manual analysis of the function body, which showed that only boolean operations were performed on certain variables.

Bounds reduction of other variables: It was sometimes possible to deduce suitable bounds for other integer–type variables through code inspection. An example is where a particular variable is only used as the input to a `switch` statement. In this case the variable bounds can be set according to the minimum and maximum values matched by the `case` statements, ensuring that the `default` statement will also be matched.

By default, AUSTIN does not include any form of input domain reduction. To ensure the comparison between the ETF and AUSTIN was as fair as possible, AUSTIN was extended to apply the same input domain reduction as the ETF which has been described above. AUSTIN's test driver was modified to ignore the same global variables as were discarded by the ETF as unused, despite the limitations of the underlying technique used in the ETF to perform such analysis. User–based input domain reductions performed in the ETF were saved in an XML file. AUSTIN was adapted to parse this file and apply the same configuration to its inputs. The modifications to AUSTIN were performed for this study only and are not part of the downloadable tool.

[1] C99 does contain a bool type.

Table 3.3: Summary of functions with a statistically significant difference in the branch coverage achieved by AUSTIN and the ETF. The columns StdDev indicate the standard deviation from the mean for ETF and AUSTIN. The t value column shows the degrees of freedom (value in brackets) and the result of the t–test. A p value of less than 0.05 means there is a statistically significant difference in the mean coverage between ETF and AUSTIN.

Function	Coverage ETF (%)	StdDev (%)	Coverage AUSTIN (%)	StdDev (%)	t value	p
02	91.15	0.09	91.81	0.42	$t(58) = 7.15$	$1.6 \cdot 10^{-9}$
08	85.53	0.00	82.89	0.00	$t(58) = Inf$	0
12	98.48	1.76	100.0	0.00	$t(63) = 4.93$	$6.3 \cdot 10^{-6}$

3.4.2 Evaluation

Effectiveness of AUSTIN. Figure 3.23 shows the level of coverage achieved by both, the ETF and AUSTIN, with error bars in each column indicating the standard error of the mean. The results provide evidence to support the claim that AUSTIN can be equally effective in achieving branch coverage than the more complex search algorithm used as part of the ETF. In order to test the hypothesis that AUSTIN is more effective than ETF in terms of branch coverage, a test for statistical significance was performed to compare the coverage achieved by each tool for each function. A two–tailed test was chosen such that *reductions* in the branch coverage achieved by AUSTIN compared with the ETF could also be tested. Since the samples were often distinctly un–normally distributed and possessed heterogeneous variances and skew, the samples were first rank–transformed as recommended by Ruxton (Rux06). Then a two–tailed t–test for unequal variances with ($p \leq 0.05$) was carried out on the ranked samples. The two–tailed t–test was used instead of the Wilcoxon–Mann–Whitney test because the latter is sensitive to differences in the shape and variance of the distributions.

As shown in Figure 3.23 and detailed in Table 3.3, AUSTIN delivered a statistically significant increase in coverage for functions 02 and 12. For function 08 AUSTIN achieved a statistically significant lower branch coverage than the ETF (82.9% vs. 85.5%, $t(58)$ = Inf). For all other functions, the null–hypothesis that the coverage achieved by the two tools comes from the same population was not rejected. Overall, these findings support and extend the previous independently obtained results by Harman and McMinn (HM09) to machine generated code.

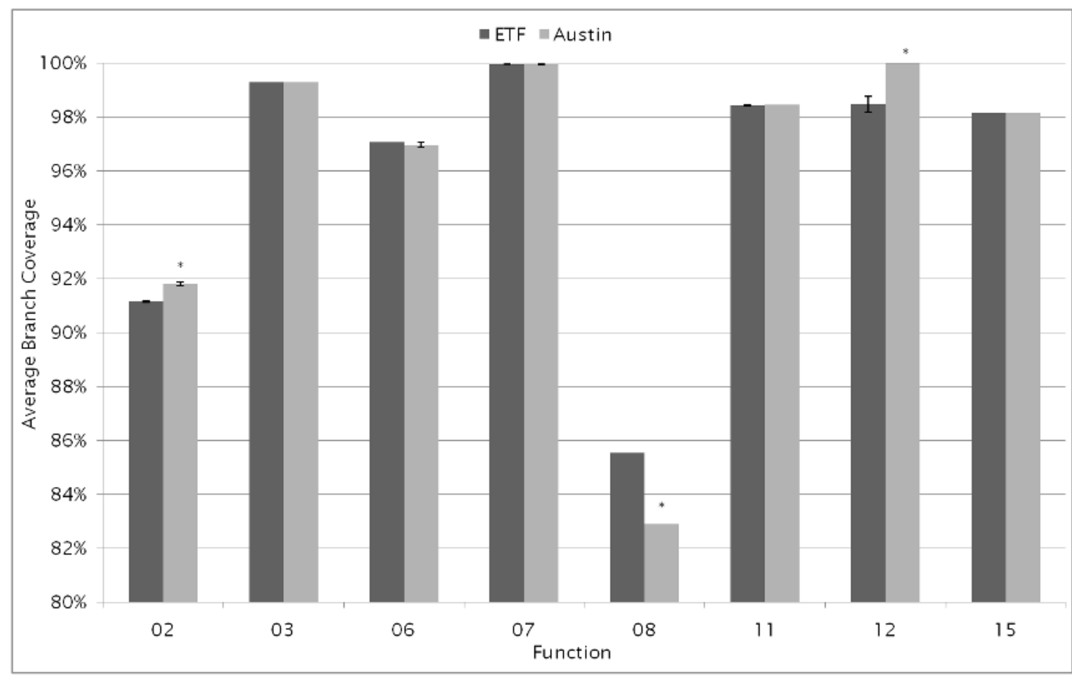


Figure 3.23: Average branch coverage of the ETF versus AUSTIN ($N \geq 30$). The $y-axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$).

Function 08 is interesting because it is the only function for which AUSTIN performs significantly worse than the ETF. Therefore the results were analysed in more detail. The first point of interest was the constant number of fitness evaluations AUSTIN used during the 30 runs of this function. Recall that AUSTIN starts with all primitive type inputs initialized to zero and all pointers set to *null*. Thus, in all 30 runs, the search starts from the same point in the search–space. Further, manual inspection of the code determined that the behaviour of function 08 is to all intents and purposes deterministic, though one cannot prove this property (without solving the halting problem). For a given solution, provided the behaviour of the program is deterministic, AUSTIN will also be deterministic, *i.e.* choose the same neighbours, up until either a random restart is required, or it has found a solution. Therefore a constant number of fitness evaluations only occurs in one of two cases: 1) AUSTIN is able to find a solution for each target branch from its initial starting point; 2) for all targets which require AUSTIN to perform a random restart, it fails to find a solution, *i.e.* the random restart has no effect on the success of AUSTIN. In this case it will continue until its fixed limit of fitness evaluations has been reached.

The second interesting result for function 08 was the significant reduction in coverage between AUSTIN and ETF. AUSTIN was unable to cover thirteen branches, which were guarded by a 'hard to cover' condition. Manual analysis of the function showed that the difficult condition becomes feasible when traversing only 2 of the 63 branches prior to it. The other 61 branches lead to a 'killing' assignment of the input variable, whose value is checked in the difficult guarding condition. The paths which contain one of the two branches, which make the difficult condition feasible, are themselves hard to cover. To see if AUSTIN's failure was due to the difficulty of the problem or, because not enough resources had been allocated, 30 runs for AUSTIN were repeated for function 08. This time no input domain reduction was used and AUSTIN was allocated a fitness budget of $100,000$ evaluations per branch. The results show that, given this larger fitness budget, AUSTIN is on average able to cover 97.60% of the branches. This is a marked increase from the average coverage of 82.89% that AUSTIN achieved with a fitness budget of only $10,000$ evaluations. The experiments could not be repeated for the ETF with the extended fitness budget of $100,000$ evaluations per branch, because the fitness budget of $10,000$ evaluations per branch is currently hard coded in the ETF.

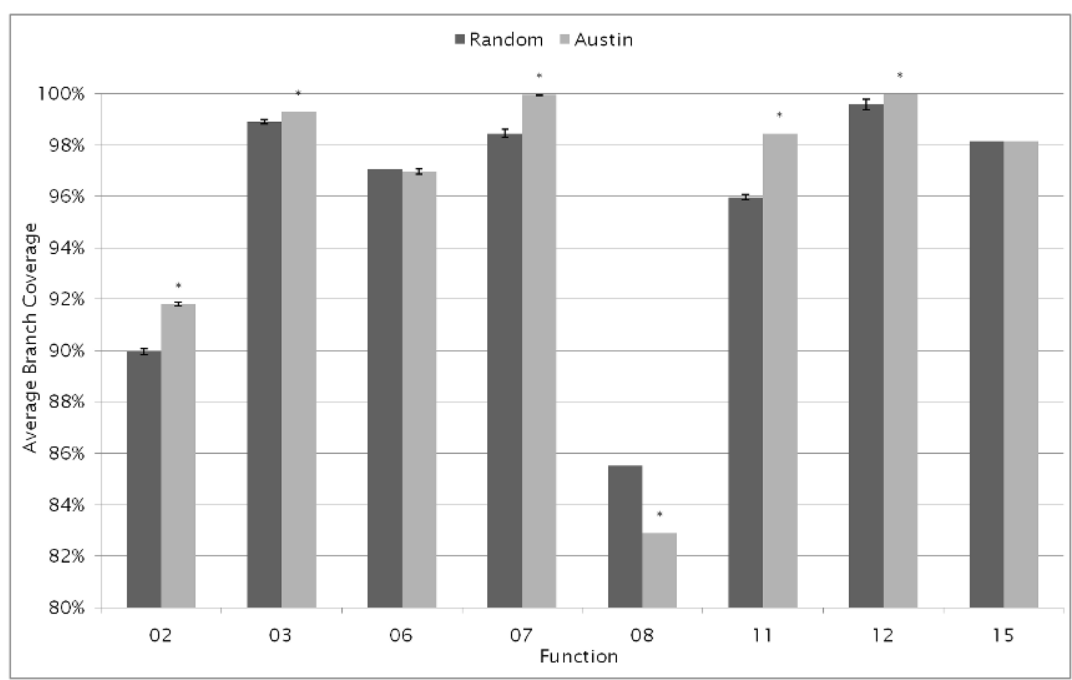


Figure 3.24: Average branch coverage of random search versus AUSTIN ($N \geq 29$). The $y - axis$ shows the coverage achieved by each tool in percent, for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean coverage ($p \leq 0.05$).

Therefore it is not possible to say how the ETF would have performed given a larger fitness budget.

Efficiency of AUSTIN. Figure 3.25 shows the average number of fitness evaluations used by both ETF and AUSTIN when trying to achieve coverage of each function. In order to test the hypothesis that AUSTIN is more efficient than ETF in terms of branch coverage, a two–tailed test for statistical significance was performed to compare the mean number of fitness evaluations used by each tool to cover each function. Since the difference in achieved coverage between the two tools was generally very small, it was neglected when comparing their efficiency. A two–tailed test was carried out to also test for functions in which AUSTIN's efficiency was worse than that of the ETF. The distributions of the samples were sufficiently normal (as determined by bootstrap re–sampling each sample–pair over 1000 iterations and visual inspection of the resulting distribution of mean values) to proceed with a two–tailed t–test for unequal variances on the raw unranked samples ($p \leq 0.05$).

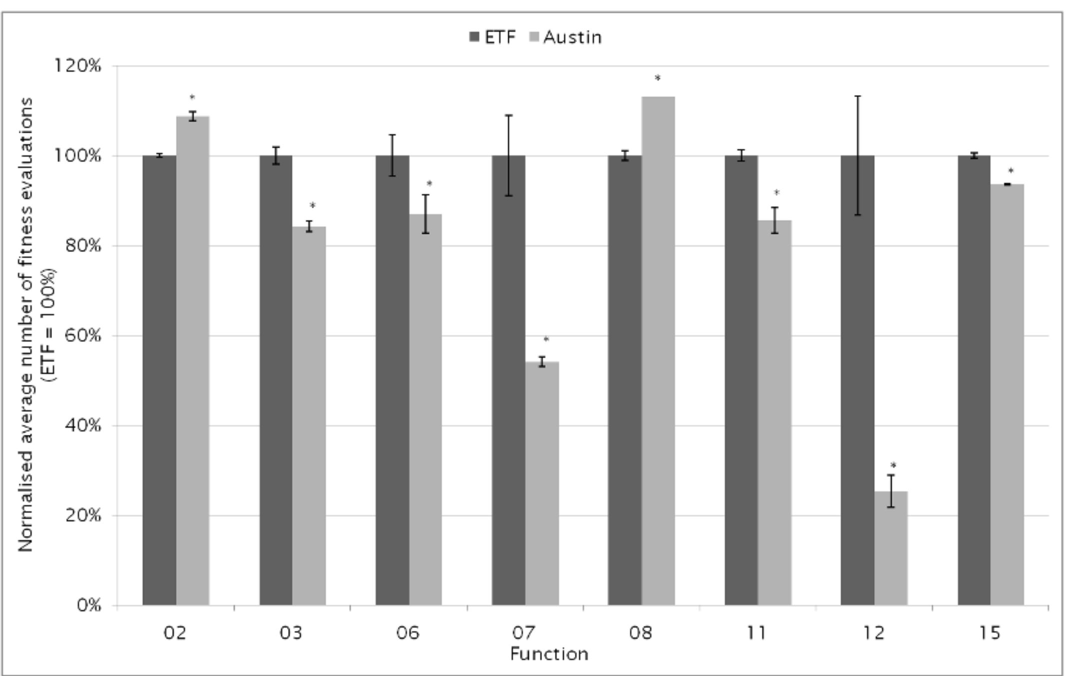


Figure 3.25: Average number of fitness evaluations (normalized) for ETF versus AUSTIN ($N \geq 30$). The $y - axis$ shows the normalized average number of fitness evaluations for each tool relative to the ETF (shown as 100%) for each of the functions shown on the $x - axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$).

Table 3.4: Summary of functions with statistically significant differences in the number of fitness evaluations used. The columns StdDev indicate the standard deviation from the mean for ETF and AUSTIN. The t value column shows the degrees of freedom (value in brackets) and the result of the t–test. A p value of less than 0.05 means there is a statistically significant difference in the mean number of fitness evaluations between ETF and AUSTIN.

Function	Evals ETF (%)	StdDev (%)	Evals AUSTIN (%)	StdDev (%)	t value	p
02	100	1.90	108.78	0.08	$t(58) = 7.67$	$2.20 \cdot 10^{-10}$
03	100	11.14	84.23	6.18	$t(58) = 7.00$	$2.00 \cdot 10^{-9}$
06	100	24.76	86.94	23.42	$t(58) = 2.10$	0.04
07	100	52.40	54.09	6.04	$t(63) = 4.79$	$1.00 \cdot 10^{-5}$
08	100	5.97	113.10	0.00	$t(58) = 9.11$	$8.70 \cdot 10^{-13}$
11	100	6.90	85.58	15.79	$t(58) = 4.41$	$4.49 \cdot 10^{-5}$
12	100	78.33	25.25	19.48	$t(63) = 5.13$	$3.02 \cdot 10^{-6}$
15	100	3.03	93.55	0.90	$t(58) = 9.22$	$5.80 \cdot 10^{-13}$

As shown in Figure 3.25, AUSTIN delivered a statistically significant increase in efficiency compared with the ETF for functions 03, 06, 07, 11, 12, 15. For functions 02 and 08, AUSTIN used a statistically significant larger number of evaluations to achieve its respective level of branch coverage. The results are summarised in Table 3.4.

Comparison with random search. As a sanity check, the efficiency and effectiveness of AUSTIN was compared with a random search. Since the random search was performed using the ETF, the same pointer handling technique and input domain reduction were applied as described in Sections 3.3 and 3.4.1 respectively. The ETF repeatedly generated and evaluated the initial random population of a standard evolutionary search. The coverage data is presented in Figure 3.24 and efficiency in Figure 3.26. Results show that, using the same tests for statistical significance as described in the previous paragraphs, AUSTIN covers statistically significantly more branches than random for functions 02, 03, 07, 11 and 12. For functions 06 and 15 there is no statistically significant difference in coverage, while for function 08, AUSTIN performs statistically significantly worse than random. However, given a larger fitness budget, as already described, AUSTIN is able to achieve a higher coverage for function 08 than the one shown in Figures 3.23 and 3.24.

Comparing AUSTIN's efficiency with that of a random search, AUSTIN is statisti-

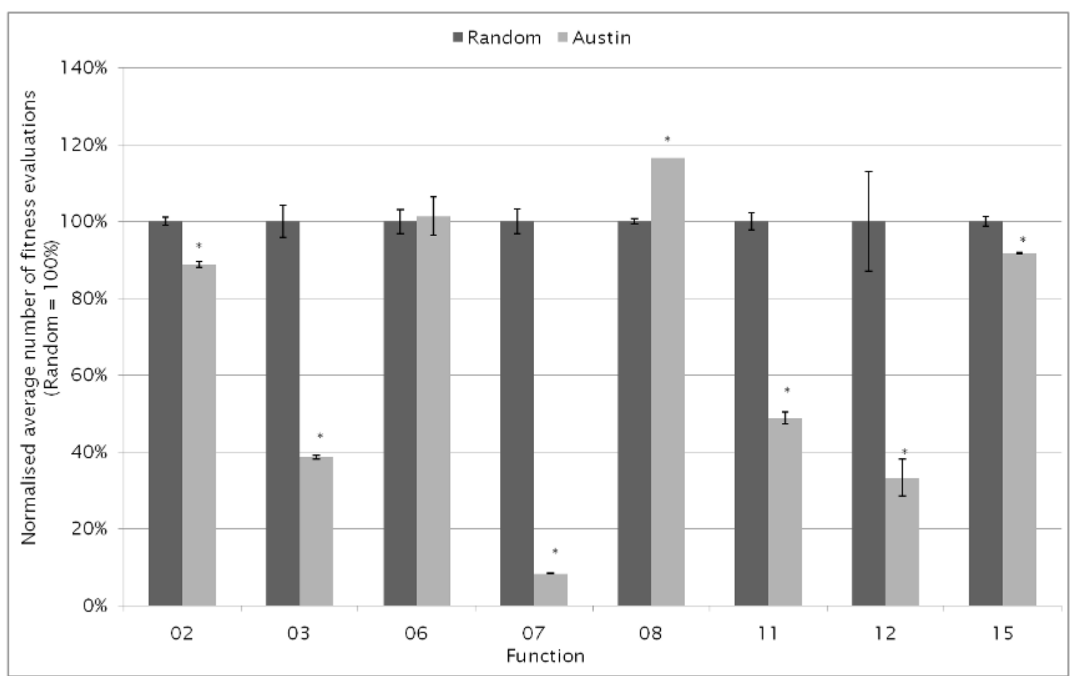


Figure 3.26: Average number of fitness evaluations (normalized) for random versus AUSTIN ($N \geq 29$). The $y-axis$ shows the normalized average number of fitness evaluations for each tool relative to the random search (shown as 100%) for each of the functions shown on the $x-axis$. The error bars show the standard error of the mean. Bars with a $*$ on top denote a statistically significant difference in the mean number of fitness evaluations ($p \leq 0.05$).

cally significantly more efficient than random for functions 02, 03, 07, 11, 12, and 15. For function 06 one cannot say that either random or AUSTIN is more efficient, while for function 08 random is statistically significantly more efficient than AUSTIN.

3.4.3 Threats to Validity

Naturally there are threats to validity in any empirical study such as this. This section provides a brief overview of the threats to validity and how they have been addressed. This chapter studied two hypotheses; 1) that AUSTIN is more effective than the ETF in achieving branch coverage of the functions under test and 2) that AUSTIN is more efficient than the ETF. Whenever comparing two different techniques, it is important to ensure that the comparison is as reliable as possible. Any bias in the experimental design that could affect the obtained results poses a threat to the *internal validity* of the

experiments. One potential source of bias comes from the settings used for each tool in the experiments, and the possibility that the setup could have favoured or harmed the performance of one or both tools.

The experiments with the ETF had already been completed as part of the EvoTest project, thus it was not possible to influence the ETF's setup. It had been manually tuned to provide the best consistent performance across the eight functions. Therefore care was taken to ensure that AUSTIN was adjusted as best as possible to use the same settings as the ETF. AUSTIN was adapted to parse ETF's XML bounds file in order to capture the input domain reduction performed by the tester during ETF's evaluation. To ensure AUSTIN discarded the same global input variables as the ETF, AUSTIN's test driver was manually adapted through analysis of the instrumented source code produced by the ETF. As can be seen from the results for function 08, the setup used may have had an adverse effect on the performance of AUSTIN. This was proven by repeating the experiments for that function with a larger fitness budget and without the input domain reduction. Those experiments showed that AUSTIN was able to achieve a much higher coverage than with its original setup.

Another potential source of bias comes from the inherent stochastic behaviour of the meta–heuristic search algorithms used in AUSTIN and the ETF. The most reliable (and widely used) technique for overcoming this source of variability is to perform tests using a sufficiently large sample of result data. In order to ensure a large sample size, experiments were repeated at least 30 times. To check if one technique is superior to the other a test for a statistically significant difference in the mean of the samples was performed. Care was taken to examine the distribution of the data first, in order to ensure the most robust statistical test was chosen to analyse the data.

A further source of bias includes the selection of the functions used in the empirical study, which could potentially affect its *external validity*, *i.e.*, the extent to which it is possible to generalise from the results obtained. The functions used in the study had been selected by Berner & Mattner Systemtechnik GmbH on the basis that they provided interesting and worthwhile candidates for automated test data generation. Particular attention was paid to the number of branches each function contained, as well as the maximum nesting level of `if` statements within a function. Finally, all the functions from the study contain machine generated code only. While the overall number of branches provides a large pool of results from which to make observations,

the number of functions itself is relatively small. Therefore, the results reported in the empirical study should not be viewed in isolation, but rather as an extension of previous work by Harman and McMinn (HM09), independently replicating their findings. Nevertheless, caution is required before making any claims as to whether these results would be observed on other functions, in particular hand written code. This concern will be addressed in Chapter 4.

3.5 Conclusion

This chapter has introduced and evaluated the AUSTIN tool for search–based software testing. AUSTIN is a fully featured, free, publicly available tool for search–based test data generation. It uses an alternating variable method, augmented with a set of simple constraint solving rules for pointer inputs to a function. Test data is generated by AUSTIN to achieve branch coverage for large C functions. In a comparison with the ETF, a state–of–the–art evolutionary testing framework, AUSTIN performed as effectively and considerably more efficiently than the ETF for eight non–trivial C functions, which were implemented using code–generation tools.

As mentioned in the Section 3.4.3, further experiments are required in order to gain a better understanding of how well AUSTIN and search–based testing in general perform on real–world code, especially hand written code. This is the subject of the next chapter.

Chapter 4

An Empirical Investigation Comparing AUSTIN and CUTE

4.1 Introduction

The previous chapter described how search–based testing has been extended to generate input values for pointers and dynamic data structures. It draws on ideas first introduced in dynamic symbolic execution, another strand of research in software testing that has seen a sustained growth of interest over recent years (GKS05; CE05; SMA05; TdH08). The majority of publicly available testing tools are also based on dynamic symbolic execution techniques. This chapter aims to evaluate the extended version of search–based testing introduced in Chapter 3 against a dynamic symbolic execution based tool, CUTE. Dynamic symbolic execution combines *con*crete execution with symb*olic* execution and is thus sometimes referred to as *concolic* testing. The term *concolic* was coined by Sen *et al.* (SMA05) in their work introducing the CUTE tool. It stands for combining **conc**rete execution with symb**olic** execution. The remainder of this chapter will use the term concolic in place of dynamic symbolic execution.

Whilst many papers argue that both concolic and search–based techniques are better than random testing (SMA05; WBS01; HM09), there has been little work investigating and comparing their effectiveness with real–world software applications. Previous work has tended to be small–scale, considering only a couple of programs, or considering only parts of individual applications to which the test generation technique is known in advance to be applicable. Furthermore, the test data generators themselves have

tended not to apply test data generation tools 'out of the box'; *i.e.* without customization, special handling for different test subjects, or parameter tuning. This leaves the literature without convincing answers to several important questions, including:

1. How effective are concolic and search–based tools when applied to real–world software applications?
2. To what types of program are the concolic and search–based approaches best suited?
3. How long does it take for concolic and search–based tools to achieve certain levels of coverage?

The aim of this chapter is to provide answers to these questions. In order for automated test data generation approaches to achieve their full potential it is necessary for them to be evaluated using realistic non–trivial programs without any 'special' (*i.e.* human) intervention to smooth over the difficult 'real–world' challenges that might be encountered. An empirical study is performed which compares a concolic tool, CUTE (SMA05), and AUSTIN. The test adequacy criterion under investigation is branch coverage. The primary contributions of this chapter are the following:

1. An empirical study which determines the level of code coverage that can be obtained using CUTE and AUSTIN on the complete source code of five open source programs. Perhaps surprisingly, the results show that only modest levels of coverage are possible at best, and there is still much work to be done to improve test data generators.
2. An empirical study investigating the wall clock time required for each tool to obtain the coverage that it does, and thus an indication of the efficiency of each individual approach.
3. An assessment, based on the empirical study, of where CUTE and AUSTIN succeeded and failed, and a discussion and detailed analysis of some of the challenges that remain for improving automated test data generators to achieve higher levels of code coverage.

The rest of the chapter is organized as follows. Section 4.2 provides the background information to the concolic tool CUTE, which the empirical study presented in this chapter uses. Section 4.3 outlines the motivation for this work, the research questions addressed, and the gap in the current literature this chapter is trying to close. The empirical study, results and answers to the research questions are presented in Section 4.4, whilst threats to validity are addressed in Section 4.5. Section 4.6 discusses remaining open problems in automated test data generation as a result of the empirical work carried out in this chapter. Section 4.7 presents related work before Section 4.8 concludes the chapter.

4.2 CUTE

As described in Chapter 2, concolic testing builds on the ideas of symbolic execution. For a given path through a program, symbolic execution involves constructing a *path condition*; a system of constraints in terms of the input variables that describe when the path will be executed. CUTE attempts to execute all feasible program paths, using a depth first strategy. The first path executed is the one, which is traversed with all zero or random inputs. The next path to be attempted is the previous path, but taking the alternative branch at the last decision statement executed in the path. The new path condition is therefore the same as the previous path condition, but with the last constraint negated, allowing for substitution of sub–expressions in the new path condition with sensible concrete values (as in the examples in Section 2.2.2). For programs with unbounded loops, CUTE may keep unfolding the body of the loop infinitely many times, as there may be an infinite number of paths. The CUTE tool is therefore equipped with a parameter which places a limit on the depth first path exploration strategy. A more detailed description of how CUTE handles input variables of both primitive and pointer type, has been presented as part of the literature review in Chapter 2.

4.3 Motivation and Research Questions

One of the first tests for any automatic test data generation technique is whether it outperforms random testing. Many authors have demonstrated that both concolic

Table 4.1: Details of the test subjects used in the empirical study. In the 'Functions' column, 'Non–trivial' refers to functions that contain branching statements. 'Top–level' is the number of non–trivial, public functions that a test driver was written for, whilst 'tested' is the number of functions that were testable by the tools (*i.e.* top–level functions and those that could be reached interprocedurally). In the 'Branches' column, 'tested' is the number of branches contained within the tested functions.

Test	Lines	Functions				Branches	
Subject	of Code	Total	Non–Trivial	Top Level	Tested	Total	Tested
libogg	2,552	68	33	32	33	290	284
plot2d	6,062	35	35	35	35	1,524	1,522
time	5,503	12	10	8	10	202	198
vi	81,572	474	399	383	405	8,950	8,372
zile	73,472	446	339	312	340	3,630	3,348
Total	169,161	1,035	816	770	823	14,596	14,084

based and search–based techniques can outperform purely random test data generation. However, there are fewer studies that have attempted to evaluate concolic and search–based approaches on real–world programs.

Previous studies have tended to be small–scale (Kor90; WBS01) or, at least in the case of search–based approaches, concentrated on small 'laboratory' programs. Where production code has been considered, work has concentrated solely on libraries (Ton04) or individual units of applications (HM09); usually with the intention of demonstrating improvements or differences between variants of the techniques themselves.

Studies involving concolic approaches have also tended to focus on illustrative examples (SMA05; WMMR05; CTS08; CE05), with relatively little work considering large scale real–world programs such as the `vim` text editor (BS08; MS07), network protocols (God07a) or windows applications (God07b). Furthermore, no studies have compared the performance of concolic and search–based testing on real–world applications.

The research questions to be answered by the empirical study are therefore as follows:

RQ 1: How effective are concolic and search–based test data generation for real–world programs? Given a set of real–world programs, how good are concolic and search–based test data generators at achieving structural coverage?

RQ 2: What is the relative efficiency of each individual approach? If it turns out that both approaches are more or less equally effective at generating test data, efficiency will be the next issue of importance as far as a practitioner is concerned. If one approach is more effective but less efficient than the other, what is the trade off that the practitioner has to consider?

RQ 3: Which types of program structure did each technique fail to cover? Which functions could the tools not handle, and for the functions which could be handled, what types of branches remained stubborn to the test generation process? Thus, what are the challenges that remain for automatic test data generation tools?

4.4 Empirical Study

The empirical study was performed on a total of 87,589 preprocessed lines of C code contained within four open source programs. This is the largest study of search–based testing by an order of magnitude and is similar in size to the largest previous study of any form of concolic testing.

4.4.1 Test Subjects

Details of the subjects of the empirical study are recorded in Table 4.1. A total of 770 functions were tested. Since the study is concerned with branch coverage, trivial functions not containing any branches were ignored. In addition, further functions had to be omitted from the study, because they could not be handled by either CUTE or AUSTIN. These included functions whose inputs were files, data structures involving function or `void` pointers, or had variable argument lists. These problems are discussed further in Section 4.4.3 in the answer to RQ 3.

The programs chosen are not trivial for automated test data generation. `libogg` is a library used by various multimedia tools and contains functions to convert to and from the `Ogg` multimedia container format, taking a bitstream as input. `plot2d` is a relatively small program which produces scatter plots directly to a compressed image file. The core of the program is written in ANSI C, however the entire application includes C++ code. Only the C part of the program was considered during testing because the tools handle only C. `time` is a GNU command line utility which takes, as input, another

process (a program) with its corresponding arguments and returns information about the resources used by a specific program, including wall–clock and CPU times. `vi` is a common text editor in Unix, and makes heavy use of string operations; as does `zile`, another GNU program, designed to be a more lightweight version of Emacs.

4.4.2 Experimental Setup

Each function of each test subject was taken in turn (hereinafter referred to as the 'FUT' – Function Under Test), with the aim of recording the level of coverage that could be achieved by each respective tool.

Since CUTE and AUSTIN take different approaches to test data generation, care had to be taken in setting up the experiments so that the results were not inadvertently biased in favour of one of the tools. The main challenge was identifying suitable stopping criteria that were 'fair' to both tools. Both tools place limits on the number of times the function under test can be called, yet this is set on a per–function basis for CUTE and a per–branch basis for AUSTIN. Furthermore, one would expect CUTE to call the function under test less often than AUSTIN because it carries out symbolic evaluation. Thus, setting a limit that was 'equal' for both tools was infeasible. Therefore it was decided that each limit would be set to a high value, with a time limit of 2 minutes of wall clock time per FUT as an additional means of deciding when a tool's test data generation process should be terminated.

CUTE's limit was set to the number of branches in the FUT multiplied by $10,000$. CUTE can reach this limit in only two cases; firstly if it keeps unfolding a loop structure, in which case it will not cover any new branches; or secondly if the limit is less than the number of interprocedural branches (which was not the case for any of the test subjects considered). AUSTIN's limit was set to $10,000$ FUT executions per branch. Often AUSTIN did not exhaust this limit but was terminated by the overall time limit instead.

Thirty 'trials' were performed for each tool and each function of each test subject. AUSTIN is stochastic in nature, using random points to restart the search strategy once the initial search, starting with all primitives as zero, fails. Thus, several runs need to be performed to sample its behaviour. CUTE was also executed thirty times for each function, so that AUSTIN did not benefit unfairly from multiple executions.

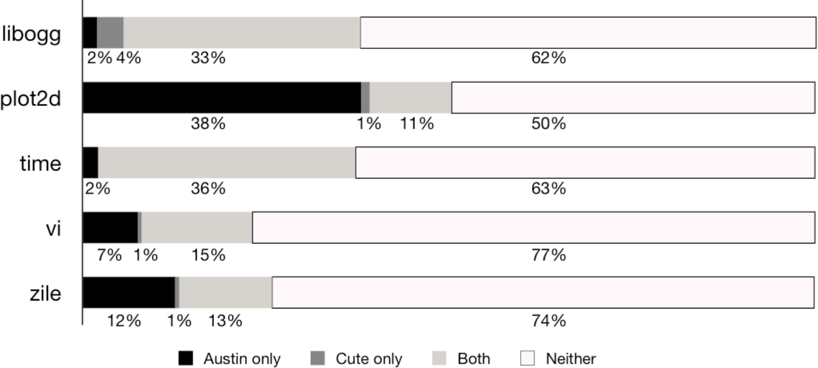
Coverage was measured in two ways. The first is respective to the branches covered in the FUT only. A branch is counted as covered if it is part of the FUT, and is executed at least once during the thirty trials. The second measure takes an interprocedural view: a branch is counted as covered if it is part of the FUT or any function reachable through the FUT. Interprocedural coverage is important for CUTE, since path conditions are computed in an interprocedural fashion. Any branches covered interprocedurally by AUSTIN, however, are done so serendipitously as the tool only explicitly targets branches in the FUT.

Apart from the settings necessary for a fair comparison, as discussed above, both tools were applied 'out of the box', *i.e.* with default parameters and without the writing of special test drivers for any of the test subjects. As mentioned in Section 4.2, CUTE has an option to limit the level of its depth first search, thus preventing an infinite unfolding of certain loops. However, as it is not generally known, *a priori*, what a reasonable restriction is, CUTE was used in its default mode with no specified limit, *i.e.* an unbounded search.

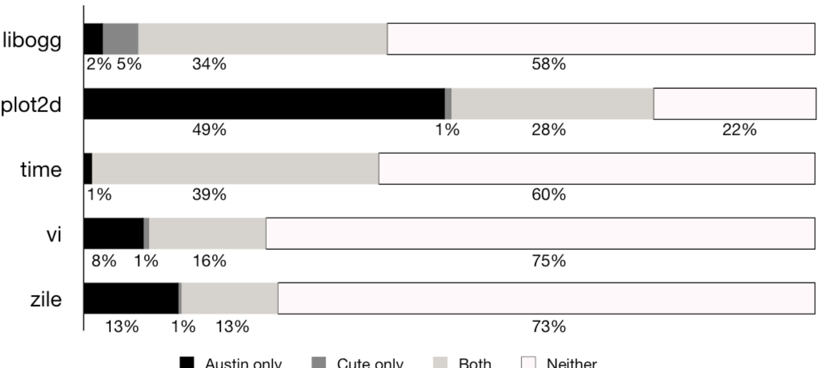
The test driver for both tools is not only responsible for initializing input parameters, but also the place to specify any preconditions for the function under test. AUSTIN generates a test driver automatically by examining the signature of the FUT. The test drivers for CUTE had to be written manually but were constructed using the same algorithm as AUSTIN. Writing preconditions for functions without access to any specifications is non–trivial. For the study only the source code was available with no other documentation. Therefore it was decided the only precondition to use was to require top level pointers (*i.e.* formal parameters or global variables of pointer type) to be non–*null*.

4.4.3 Answers to Research Questions

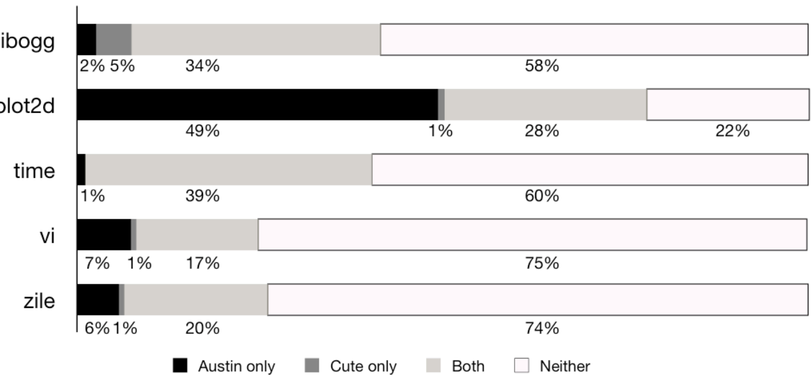
RQ 1: How effective are concolic and search–based test data generation for real–world programs? Figure 4.1 plots three different 'views' of the coverage levels obtained by CUTE and AUSTIN with the test subjects. The first view, Figure 4.1(a), presents coverage of branches in the FUT only. However, CUTE places an emphasis on branches covered in functions called by the FUT, building up path conditions interprocedurally. For AUSTIN interprocedural branch coverage is incidental, with test



(a) Branches covered only as part of the function under test



(b) Branches covered in the function under test and interprocedurally



(c) Branches covered in functions that CUTE can handle only

Figure 4.1: Branch coverage for the test subjects with CUTE and AUSTIN. CUTE explicitly explores functions called by the function under test, whereas AUSTIN does not. Therefore the graph 4.1(a) counts only branches covered in each function tested individually. Graph 4.1(b) counts branches covered in the function under test and branches covered in any functions called. Graph 4.1(c) is graph 4.1(b) but with certain functions that CUTE cannot handle excluded.

Table 4.2: Test subjects used for comparing wall clock time and interprocedural branch coverage.

Test Subject	Function	Branches	
		Function Under Test	(Interprocedural)
libogg	ogg_stream_clear	8	(8)
	oggpack_read	14	(14)
plot2d	CPLOT_BYTE_MTX_Fill	4	(8)
	CPLOT_DrawDashedLine	56	(56)
	CPLOT_DrawPoint	16	(16)
time	resuse_end	6	(6)
vi	_compile	348	(348)
	exitex	4	(4)
	main_old	152	(174)
	plod	174	(174)
	pofix	4	(4)
	vappend	134	(426)
	vgetline	220	(228)
	vmain	404	(480)
	vmove	30	(30)
	vnpins	6	(108)
	vputchar	148	(212)
zile	astr_rfind_cstr	6	(6)
	check_case	6	(6)
	expand_path	82	(84)
	find_window	20	(20)
	line_beginning_position	8	(8)
	setcase_word	40	(74)
Total		1,890	2,494

Table 4.3: Interprocedural branch coverage for the sample branches from Table 4.2 and the wall clock time taken to achieve the respective levels of coverage.

Function	CUTE			AUSTIN		
	Time (s)	Branches covered Function	(Inter-procedural)	Time (s)	Branches covered Function	(Inter-procedural)
libogg						
ogg_stream_clear	0.84	7	(10)	134.75	5	(7)
oggpack_read	0.24	2	(2)	0.18	2	(2)
plot2d						
CPLOT_BYTE_MTX_Fill	131.25	4	(4)	130.05	1	(1)
CPLOT_DrawDashedLine	130.43	13	(13)	130.4	37	(37)
CPLOT_DrawPoint	0.51	13	(13)	131.82	13	(13)
time						
resuse_end	0.42	4	(4)	131.84	5	(5)
vi						
_compile	2.11	26	(26)	34.14	24	(24)
exitex	146.47	1	(1)	0.22	1	(1)
main_old	0.45	0	(0)	0.31	0	(0)
plod	1.58	18	(18)	137.17	18	(18)
pofix	0.41	1	(1)	135.08	1	(1)
vappend	0.53	6	(6)	0.26	6	(6)
vgetline	0.81	3	(3)	0.26	3	(3)
vmain	0.27	3	(3)	0.3	3	(3)
vmove	0.79	3	(3)	0.24	3	(3)
vnpins	0.22	2	(2)	0.25	2	(2)
vputchar	0.2	4	(4)	0.25	4	(4)
zile						
astr_rfind_cstr	0.45	2	(2)	0.17	2	(2)
check_case	0.38	1	(1)	130.49	6	(6)
expand_path	0.37	0	(1)	0.16	0	(1)
find_window	0.4	1	(1)	131.4	1	(1)
line_beginning_position	0.27	0	(0)	0.18	0	(0)
setcase_word	0.31	0	(0)	0.18	0	(0)

generation directed only at the FUT. Therefore, Figure 4.1(b) plots interprocedural coverage data which, in theory, should be favourable to CUTE. Finally, CUTE could not attempt 138 functions, containing 1,740 branches. Some of these functions involved `void` pointers, which cannot be handled by CUTE. However, a number of functions could not be tested by CUTE because the test subject did not compile after CUTE's instrumentation. For certain functions of the `zile` test subject, the instrumentation casts a data structure to an unsigned integer, and subsequently tries to dereference a member of the data structure, which results in an error. Since CUTE's exploration behaviour is interprocedural, all functions within this source file became untestable. Thus, Figure 4.1(c) plots interprocedural branch coverage, but removing these branches from consideration.

Strikingly, all three views of the coverage data show that in most cases, the majority of branches for an application were not covered by *either* tool. The only exception is the `plot2d` test subject. Here, AUSTIN managed 77% coverage taking interprocedural branches into account, compared to CUTE's 50%. Code inspection revealed that 13 functions in `plot2d` contained unbounded loops. CUTE therefore never attempted to cover any more branches preceding the body of the loop (both intraprocedural and interprocedural) and instead kept increasing the number of loop iterations by one until its timeout or iteration limit was reached. For all other subjects, coverage for either tool does not exceed 50% whatever 'view' of the data is considered. It has to be noted, however, that AUSTIN does seem to cover a higher number of the branches. When a modified path condition falls outside the supported theory of CUTE's constraint solver, unlike AUSTIN, CUTE does not try a fixed number of random 'guesses' in order to find test data. AUSTIN on the other hand will spend 10,000 attempts at covering the branch. In the worst–case this is equal to performing a random search. Nevertheless, it has a higher chance of finding test data than CUTE simply because it spends more effort per branch.

To conclude, these data suggest that the concolic and search–based approaches are not as effective for real–world programs as researchers may have previously been led to believe by previous smaller–scale studies. RQ 3 aims to identify where the problems were and where the challenges remain. Before this, the efficiency of each of the tools is examined with respect to a subset of the branches.

```
void testme(int upb, ...) {
  ...
  while(upb > 0) {
    upb = upb - 1;
    ...
  }
  ...
}
```

Figure 4.2: An example used to illustrate how a random restart in AUSTIN can affect the runtime of a function under test, and thus the test data generation process. Very large values of `upb` may have a significant impact on the wall clock time of the test data generation process.

RQ 2: What is the relative efficiency of both approaches? In order to answer this research question, a random sample of 23 functions was taken and the performance of each individual tool analysed further. These functions are listed in Table 4.3 and comprise 1,890 branches, with a further 604 reachable interprocedurally.

CUTE times out (reaching the 120 second limit) on three occasions. This is because CUTE gets stuck unfolding loops in functions called interprocedurally. AUSTIN times out on eight occasions. For example, the function `ogg_stream_clear` from `libogg` takes as input a pointer to a data structure containing 18 members, one of which is an array of 282 unsigned characters, while 3 more are pointers to primitive types. Since AUSTIN does not use any input domain reduction, it has to continuously cycle through a large input vector in order to establish a direction for the search so it can start applying its pattern moves. Secondly, unbounded loops cause problems for AUSTIN with respect to the imposed timeout. Whenever AUSTIN performs a random restart, it has a high chance of increasing the number of loop iterations by assigning a large value to the termination criterion, thus slowing down execution time of the function under test. An example is shown in Figure 4.2. Depending on the value of `upb`, the function will take varying amounts of time to execute.

The table does not reveal a prevailing pattern that would allow us to simply conclude that 'CUTE is more efficient' or vice versa. The results are very much dependent on the function under consideration. However, unless the tool gets 'trapped' (*e.g.* CUTE

Table 4.4: Errors encountered for test data generation for the sample of branches listed in Table 4.3.

	CUTE	AUSTIN
Timeout	91	277
Aborted	0	113
Segmentation fault	419	180
Floating point error	60	90
Failed to take predicted path	30	*n/a*

continually unfolding a loop), each tool can be expected to terminate within a second, which is an entirely practical duration.

RQ 3: Which types of program structure did each technique fail to cover?

Table 4.4 logs common reasons why either tool terminated abnormally for the sample of 23 functions recorded Table 4.3.

Segmentation faults. CUTE terminated prematurely 419 times, and AUSTIN 180 times because of segmentation faults. These faults are the result of implicit constraints on pointers (*i.e.* missing guarding statements) and a consequence of assigning 'bad' values to input parameters. Consider the example below (taken from `libogg`). The variable `vals` is an indication of the number of elements in the array `b`. If this relationship is not observed during testing, *e.g.* by assigning *null* to `b` and setting `vals` > 0, the program will crash with a segmentation fault.

```
void
cliptest(unsigned long *b,int vals,...){ ...
 for(i=0;i<vals;i++)
    oggpack_write(&o,b[i],bits?bits:ilog(b[i]));
 ... }
```

Floating point exceptions. Another cause of system crashes for both tools were floating point exceptions. The IEEE 754 standard classifies 5 different types of exceptions: invalid operation; division by zero; overflow; underflow; inexact calculation.

During the sample study CUTE raised 60 floating point exceptions and AUSTIN 90. The signal received for both tools was `SIGFPE`, which terminates the running process. Code inspection revealed that in all cases the cause of the exception was a division by

zero error. This is not surprising since all primitive inputs are initialized to zero by both tools.

Unhandleable functions. Out of all functions considered during the main study, 46 could not be tested by both tools and an additional 138 not by CUTE because of unhandled input types. AUSTIN, for example, initializes void pointers to *null* and does not attempt to assign any other value to them. CUTE cannot handle such pointers at all, and generates an undefined `_cu__unhandledInputType` call. It is not clear what would be a good strategy for setting such pointers. Source code analysis may be required to determine the possible data types.

A similar problem to void pointers can be observed with the `va_list` type, allowing variable length argument functions in C. This type is essentially a linked list container, and the data types of the data items cannot always be established a priori. An implementation issue is also related to variable argument lists. A call to `__builtin_va_arg` in the version of the `glibc` library, takes the data type:

```
__builtin_va_arg(marker, mytype)
```

as its second parameter. CIL transforms such calls into an internal representation of

```
__builtin_va_arg(marker, sizeof(mytype), &x);
```

where `x` is a variable of `mytype`. When printing the source code from CIL, the printer will try to print the original code. Due to the code simplification transformations used, the internal CIL form is printed instead, causing a compilation error.

Out of all available functions in the preprocessed C code, 19 had a function pointer as one of their formal parameters. This is another input type that causes problems for both CUTE and AUSTIN. Currently the tools lack the infrastructure to either select a matching function from the source code, or, to generate a stub which matches the function signature. One way to address the problem would be to manually supply the address of a function, but then the test data generation process would not be fully automated. As mentioned in Chapter 3, AUSTIN assigns *null* to all function pointers, which will of course crash the program as soon as the application attempts to use the function pointer.

Many real–world programs contain some form of file I/O operation. Yet neither tool fully supports functions whose parameter list contains a pointer to a file structure. During this study, the `FILE` data structure contained incomplete type information in the accessible source code, and thus neither tool had the information required to instantiate such inputs. Even if all the type information had been available, the tools would not have been able to generate 'valid' inputs. This is because each member of the data structure has a semantic meaning. Therefore it would not make sense to populate it with random values. The same applies to any `_IO_FILE` data structure (*e.g.* as used by `stderr` or `stdout`). Populating these with arbitrary values will likely result in nothing more than program crashes, or extensive testing of error handling procedures.

4.4.3.1 CUTE–specific issues

During the study, CUTE failed to cover branches for the following reasons. As explained in the experimental setup, only the source file containing the current function under test was instrumented. Any other source files of the program remained untouched, and thus formed a 'black box' library as far as CUTE was concerned. Black box functions (system or application specific) influence CUTE's effectiveness in two ways.

Return values from black box functions cannot be represented symbolically, and are thus treated as constants in a path condition where applicable. The underlying assumption is that their value remains unchanged over successive iterations with similar inputs. If this is not the case, successive executions may not follow the sub–path predicted by CUTE (which results in CUTE aborting its current search, reporting '... CUTE failed to take the predicted path', and restarting afresh).

Worse still, formal parameters of the function under test may be passed by reference to a black box function which, in turn, modifies their value. Since CUTE cannot be aware of how these inputs may change, it is likely that solutions generated by CUTE do not satisfy the real path conditions and thus also result in prediction errors by CUTE.

An illustrative example can be found in the `time` program, involving the function `gettimeofday`, which populates a data structure supplied as input parameter to the function. While CUTE attempts to set members of this structure in order to evaluate the predicate shown below with a true outcome, any values of the data structure are overridden before reaching the predicate. A weakness of CUTE is that it fails to recognize the indirect assignment to the input data structure via the function `gettimeofday`.

A better approach would be to assume any pointer passed to an uninstrumented piece of code may be used to update blocks of memory reachable through that pointer.

```
resuse_end (pid_t pid, RESUSE *resp){...
 gettimeofday (&resp->elapsed,(struct timezone *) 0);
 ...
 if (resp->elapsed.tv_usec < resp->start.tv_usec)
  //target }
```

One of the acclaimed strengths of concolic testing is its ability to overcome weaknesses in constraint solvers by simplifying symbolic expressions so they can be handled through existing theories. The light–weight solver used by CUTE requires many such simplifications. In fact CUTE only handles additions, subtractions, (side effect free) assignments an linear multiplication statements symbolically. Non–linear expressions are approximated as described in Section 4.2, while all other statements (*e.g.* divisions) are simply ignored (in symbolic terms). Inspection of the source code for the different programs revealed that path conditions become very quickly dominated by 'constant' values due to the nature of operations used in the code. The lack of symbolic information means inverted constraints quickly become infeasible within the given path condition. Thus CUTE is restricted in the number of execution paths it can explore.

Another implementation related drawback for CUTE is its inability to test code containing bitfields in data structures. As part of its instrumentation, CUTE attempt to take the address of a bitfield whenever it is used. This is, of course, not possible and results in the `gcc` compilation error 'cannot take address of bit-field ...', as described in Section 3.4.

Finally CUTE failed to cover certain branches because of an unbounded depth first path exploration strategy (CUTE's default mode), which resulted in CUTE unfolding unbounded loops, without ever attempting to cover any alternative branches in the code. Consider the example below, taken from the `plot2d` program. The first predicate `if( to_x - x == 0 )` is satisfied by default (all primitive inputs are instantiated with equal values). The unbounded loop contained within the `if` statement ensures CUTE has an unlimited number of paths to explore before returning. Any code following the `if` statement is never explored, even though it makes up the bulk of the function.

```
BOOL CPLOT_DrawDashedLine(...,const int x, ...
  const int to_x,
  const int to_y, ...) {...
  if( to_x - x == 0 ) {
    for( row=y; row<=to_y; row++ ){ ... }
    return TRUE; }
  //CODE NEVER EXECUTED BY CUTE
}
```

The function `exitex` highlighted an interesting behaviour in CUTE. Its input space consists of two integer variables, one a global parameter and the other a formal parameter. The body of the function is shown below.

```
int exitex(int i){
 if (failed != 0 && i == 0)
  i = failed;
 _exit(i);
 return 0;
}
```

In the first iteration, both `i` and `failed` will be 0. This means the function `_exit` is called with a value of 0, indicating successful termination of the program. The `_exit` function terminates the running process and exits with the supplied status code. Since it indicates no error, it would not get caught by any registered signal handlers. CUTE however terminates before it can save its branch history (of what has been covered) to a file for subsequent iterations. CUTE's constraint solver can easily find a solution to the constraint `failed != 0 && i == 0` and thus CUTE is left with an infinite amount of feasible branches and continues to run until its iteration limit is reached.

4.4.3.2 AUSTIN–specific issues

Search–based testing is most effective when the fitness landscape used by an algorithm provides ubiquitous guidance towards the global (maximum) minimum. When a landscape contains plateaus, a guided search is often reduced to a random search in an attempt to leave a flat fitness region. The 'shape' of the fitness landscape primarily depends on the fitness function used to evaluate candidate solutions (test cases).

Data dependencies between variables are a major contributor to plateaus in a fitness landscape, best illustrated by the flag problem (HHH^{+}04). While work has been done

to tackle this problem, no standardized approach exists for including such information in the fitness computation. The fitness function used by AUSTIN does not explicitly consider any data dependencies either.

The test subjects used during the empirical study contained many predicates exhibiting flag like properties, introducing spikes and plateaus into the fitness landscape. Consider the example shown below (taken from `plot2d`), where the predicate depends on a function returned flag. When a flag only has relatively few input values which make it adopt one of its two possible values, it will be hard to find such a value (WBW07).

```
BOOL
CPLOT_BYTE_MTX_isNull(CPLOT_structByteMatrix *M)
{
  if( !M )
    return (1);
  if( M->data == ((void *)0) )
    return (1);
  return (0);
}

BOOL
CPLOT_BYTE_MTX_Fill(CPLOT_structByteMatrix *dst,
{
   ...
   if( CPLOT_BYTE_MTX_isNull( dst ) )
      return 0;

//TARGET BRANCHES NEVER EXECUTED BY AUSTIN
}
```

For the above example, a random strategy of setting the input (and its members) to *null* or non–*null* will have a good chance of reaching the bulk of the body from `CPLOT_BYTE_MTX_Fill`. The code snippet was chosen not to illustrate the problem of function assigned flags, but to highlight another problem in AUSTIN's strategy. While symbolic information is collected across function calls, AUSTIN only computes intraprocedural control flow information. In order to take the false branch of the first `if` statement in `CPLOT_BYTE_MTX_Fill`, execution needs to follow the false branches of both conditionals in `CPLOT_BYTE_MTX_isNull`. The false branch shown in `CPLOT_BYTE_MTX_Fill` is not control dependent on the predicates in `CPLOT_BYTE_MTX_isNull`,

and thus AUSTIN will not attempt to modify `dst`. AUSTIN's pointer rules can only be applied to critical branching nodes.

During the sample study, AUSTIN reached its runtime limit 10 times while attempting coverage of a function. No input domain reduction was used, and thus at times AUSTIN had to optimize large input vectors. In the worst–case, it has to execute a function *at least* $2 * n + 1$ times to cover a target branch, where n is the size of the input vector. Previous work (HHL[+]07) has shown that reducing the search–space for an AVM can statistically significantly improve the efficiency of the search. When efficiency and a runtime limit are linked, it impacts the effectiveness of the search.

AUSTIN reported an abnormal termination of the program in 113 runs during the sample study. This does not mean having executed the `abort` function, but rather that the program under test returned an unrecognised error code. 23 of these were caused by passing an unrecognised status code to an `exit` function, propagated through by inputs from the function under test. The remainder were triggered by `longjmperror`, which aborts a program. An execution environment is saved into a buffer by a call to `setjmp`. The `longjmp` command enables a program to return to a previously saved environment. The `longjmperror` is raised if the environment has been corrupted or already returned.

4.5 Threats to Validity

Attempting to compare two different approaches faces a number of challenges, thus it is important to ensure that the comparison is as fair as possible. The study presented here compares two widely studied approaches to test data generation, and also seeks to explore their applicability to real–world code. As was the case with the study presented in Chapter 3, this study also faces a number of threats to the validity of the findings.

The first issue to re–address is that of the internal validity of the experiments. As with the previous empirical study, a potential source of bias comes from the settings used for each tool in the experiments, and the possibility that the setup could favour or harm the performance of either tool. In order to address this, default settings were used where possible. Where there was no obvious default (*e.g.* termination criteria), care was taken to ensure that reasonable values were used, and that they allowed a

sensible comparison between performance of both tools. To address any bias from the stochastic behaviour of AUSTIN, experiments in this study were also repeated 30 times.

A further source of bias includes the selection of the programs used in the empirical study, because these potentially affect external validity. The rich and diverse nature of programs makes it impossible to cover all possible characteristics of programs. However, where possible, a variety of programming styles and sources have been used. The study draws upon code from real–world open source programs. It should also be noted that the empirical study drew on over 700 functions comprising over 14,000 branches, providing a large pool of results from which to make observations.

The data were collected and analysed in three different ways: taking into account coverage in the function under test only, interprocedural coverage and removing functions that CUTE could not handle from the sample. No matter which analysis was conducted, the results always showed a consistently poor level of coverage. Nevertheless, caution is required before making any claims as to whether these results would be observed on other programs, possibly from different sources and in different programming languages. As with all empirical experiments in software engineering, further experiments are required in order to replicate the results here.

4.6 Discussion: Open Problems in Automated Test Data Generation

This section serves to highlight some of the open issues in automated test data generation techniques that were revealed by the empirical study.

A general concern with search–based structural testing methods is that the majority only consider control flow information as part of their fitness function. This often leads to unfortunate fitness landscapes with plateau–like features and thus to a deterioration in the search. Previous work (FK96; MH06) has addressed this problem to an extent, however no study has yet been carried out to examine the scalability of these methods in practice.

A more pressing problem for any symbolic execution based approach is the absence of explicit constraints in the source code. Constraints on input parameters often only exist as implicit preconditions to functions. While this was a serious problem for CUTE and AUSTIN during the empirical study (due to the level of segmentation faults), other

tools such as Pex (TdH08) are able to infer such conditions to an extent (and in certain cases even suggest code improvements) to avoid exceptions preventing further code exploration.

Another practical issue, mainly for CUTE, is that real programs are spread over multiple source files. While it is possible to merge source files into one file for instrumentation, this may be infeasible for very large files. Had the source files been merged for this study, it is possible that some of the 'failed to take the predicted path' errors could have been avoided. Of course many real–world applications contain a mix of programming languages and make extensive use of libraries for which the source code may not always be available. Hence, even when merging all C files, there are likely to remain uninstrumented files, causing the same problems for source code based tools as previously described.

Finally, the challenge on how to test programs like `time` or `valgrind`, which essentially take other programs as their input, remains as yet unsolved. Looking at the `main` function of either of these, it simply takes an integer (`argc`) and a pointer to a character array (`argv* []`) as inputs. Ideally one would like to describe the semantics of the inputs in some way such that it aids automated test data generation. This would also increase the likelihood of the generated test data being 'useful' rather than producing an invalid test case, *e.g.* with respect to the operational profile of a system. This is somewhat similar to the problem of generating a correct `_IO_FILE` structure, or any file type for that matter, *e.g.* valid `mp3` files *etc.*

4.7 Related Work

This section briefly reviews the relevant literature for the work presented in this chapter. It concentrates on empirical studies performed in the area of dynamic symbolic execution and search–based testing.

Jacob *et al.* (BS08) considered different search strategies to explore program paths in concolic testing and evaluated their findings on large open source applications including the Siemens benchmark suite (HR), `grep` (gre), a search utility based on regular expressions, and `vim` (VIM), a common text editor. An extended version of CUTE (MS07) has also been applied to the `vim` editor. Since its introduction, DART has been further developed and used in conjunction with other techniques to test functions from

real–world programs in an order of magnitude of 10,500 LOC (CG06; God07a). Concolic testing has also been used to search for security vulnerabilities in large Microsoft applications as part of the SAGE tool (God07b). As mentioned, studies in search–based software testing have largely involved small laboratory programs, with experiments designed to show that search–based testing is more effective than random testing (MMS01; WBS01).

This chapter complements and extends previous work. It is the first to compare both approaches on the same set of unadulterated, non–trivial, real–world test subjects. It is also the largest study of search–based testing by an order of magnitude.

4.8 Conclusion

This chapter investigated the performance of two approaches to automated structural test data generation: the concolic approach embodied in the CUTE tool, and the search–based approach implemented in the AUSTIN tool. The empirical study centred on five complete open source applications. The results show that there are many challenges remaining in making automatic test data generation tools robust and to a standard that could be considered 'industrial–strength'. This is because with the exception of one of the test subjects chosen, neither tool managed to generate test data for over 50% of the branches in each application's code.

One of the problems that emerged from the previous two chapters for AUSTIN (and search–based testing in general) is the lack of data flow information captured in the fitness functions for the branch coverage adequacy criterion. As a result, the fitness landscape is often flat, *i.e.* contains plateaus, with little or no guidance for the search. Flag variables, variables that hold one of two discrete values, `true` or `false`, represent the worst–case scenario in terms of the fitness landscape. A special type of flag variables are loop–assigned flags: flag variables that are assigned inside the body of a loop and whose use is outside the loop. These flags are the focus of the next chapter.

Chapter 5

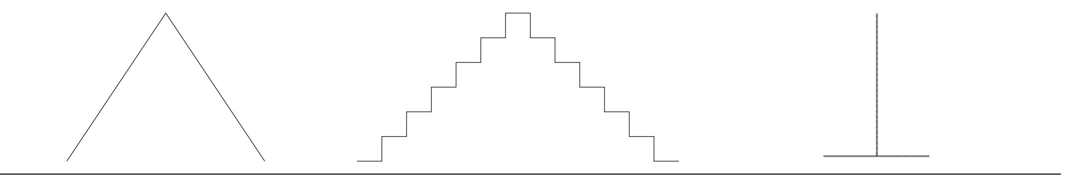
Testability Transformation

5.1 Introduction

Although search–based testing works well in many situations, provided a tester has created adequate test drivers to prevent segmentation faults and other problems highlighted in the previous chapter, it is hampered by the presence of flag variables: variables that hold one of two discrete values. Flags were one of the major contributors to the poor performance of AUSTIN as discussed in Section 4.6. Flag variables are common in embedded systems such as engine controllers, which typically make extensive use of flag variables to record state information concerning devices. Embedded systems can therefore present problems for automated test data generation. This is important, because generating such test data manually is prohibitively expensive, but the test data is required by many testing standards (Bri98; Rad92).

The flag problem is best understood in terms of the *fitness landscape*. A fitness landscape is a metaphor for the 'shape' of the hyper–surface produced by the fitness function. In the 2d case, the position of a point along the horizontal axis is determined by a candidate solution (*i.e.* an input to the program) and the height of a point along the vertical axis is determined by the computed fitness value for this input. Using the fitness landscape metaphor, it becomes possible to speak of landscape characteristics such as plateaus and gradients.

As illustrated on the right side of Figure 5.1, the use of flag variables leads to a degenerate fitness landscape with a single, often narrow, super–fit plateau and a single super–unfit plateau. These correspond to the two possible values of the flag variable.



Best case	**Acceptable case**	**Worst case**
Smooth landscape with *ubiquitous* guidance toward global optimum.	Rugged landscape with *some* guidance toward global optimum.	Dual plateau landscape with *no* guidance toward global optimum.

Figure 5.1: This figure uses three fitness landscapes to illustrate the effect flag variables have on a fitness landscape, and the resulting 'needle in a haystack' problem. The figure has been taken from the paper by Binkley *et al.* (BHLar).

This landscape is well–known to be a problem for many search–based techniques; the search essentially becomes a random search for the 'needle in a haystack' (BS03; Bot02; FK96; HHH$^+$04).

This chapter presents an algorithm for transforming programs containing loop–assigned flag variables, which cannot be handled by previous approaches. The result of the transformation is a tailored version of a program that allows existing approaches to compute representative fitness values for candidate solutions at a particular flag–controlled branch. It uses a testability transformation (HHH$^+$04): a form of transformation in which functional equivalence need not be preserved, but in which test set adequacy is preserved. The primary contributions of this chapter are as follows:

1. A testability transformation algorithm that can handle flags assigned in loops is described.

2. Results of two empirical studies evaluating the algorithm are reported. They show that the approach reduces test effort and increases test effectiveness. The results also indicate that the approach scales well as the size of the search–space increases.

The rest of this chapter is organised as follows. Section 5.2 provides an overview of background information on the flag problem and testability transformation. Section 5.3 introduces the flag replacement algorithm and Section 5.4 outlines how it has been implemented. Section 5.5 presents an empirical study which demonstrates that the

approach improves both test generation effort and coverage achieved and explores the performance of the approach as the size of the search–space increases. Section 5.6 delves into the relevant literature for the flag problem and Section 5.7 concludes the chapter.

5.2 Background

This section briefly explains the flag problem and the general characteristics of the testability transformation solution proposed. In this chapter, a flag variable will be deemed to be any variable that takes on one of two discrete values. Boolean variables are used in the examples.

5.2.1 The Flag Problem

The flag problem deals with the situation where there are relatively few input values (from some set S) that make the flag adopt one of its two possible values. This problem typically occurs with internal flag variables, where the input state space is reduced, with relatively few 'special values' from S being mapped to one of the two possible outcomes and all others being mapped to the other of the two possible flag values. As explained below, the flag problem is the hardest of what is commonly known as the internal variable problem in automated test data generation.

Consider a predicate that tests a single flag variable (*e.g.* `if( flag )`). The fitness function for such a predicate yields one of two values: either maximal fitness (for 'special values') or minimal fitness (for any other value). As illustrated in the right of Figure 5.1, the landscape induced by such a fitness function provides the search with no guidance.

A similar problem is observed with any n–valued enumeration type, whose fitness landscape is determined by n discrete values. The boolean type (where $n = 2$) is the worst–case. As n becomes larger the program becomes increasingly more testable: provided there is an ordering on the set of n elements, the landscape becomes progressively smoother as the value of n increases.

The problem of flag variables is particularly acute where a flag is assigned a value inside a loop and is subsequently tested outside the loop. In this situation, the fitness function computed at the test outside the loop may depend upon values of 'partial fitness' computed at each and every iteration of the loop. Previous approaches to

handling flags break down in the presence of such loop–assigned flags (BS03; Bot02; HHH$^+$04).

5.2.2 Testability Transformation

A testability transformation (HHH$^+$04) is a source–to–source program transformation that seeks to improve the performance of a previously chosen test data generation technique by increasing a program's 'testability'. A definition of the term testability has been attempted by several authors. Friedman and Voas (FV95) define it as "a software metric that refers to the ease with which some formal or informal testing criteria can be satisfied". In (VM95) the authors consider testability to be defined by the question "what is the probability that a code will fail if it is faulty".

This thesis follows the definition of Friedman and Voas. It considers a program's testability to be a metric describing the ease with which automated search–based test data generation techniques can successfully be applied. Empirical studies have shown that poor testability of a program is, amongst others, caused by certain programming styles (MHH01; BHLar). Identifying such programming styles in a code might then provide some information about whether a program's testability can be improved. Applying a transformation that removes problem code from a program seems a logical solution to the problem.

In traditional program transformations, the transformed program is used to replace the original program, thus one has to address issues related to functional equivalence, a demanding task since functional equivalence is undecidable. By contrast, testability transformations are only used to generate test data. Once test cases have been generated, the transformed program is of no further use. This means the burden on testability transformations is much less. For example, in order to cover a chosen branch, it is only required that the transformation preserves the set of test–adequate inputs. That is, the transformed program must be guaranteed to execute the desired branch under the same initial conditions as the untransformed program.

5.3 The Flag Replacement Algorithm

The aim of the replacement algorithm is to substitute the use of a flag variable with a condition that provides a smoother landscape. Prior work with flag variables requires

that assignments reaching a use do not occur within a loop (BS03; Bot02; HHH+04). By contrast, the algorithm presented in this paper handles flags assigned inside a loop. It does this by introducing two new real valued variables, `fitness` and `counter`. These variables replace the predicate use of a flag with an expression that supports a distance based calculation (*e.g.* `if(counter == fitness)`) to be used.

The addition of these variables is a form of instrumentation. The variable `counter` is an induction variable added to count the number of assignments to a flag in all loop iterations. The variable `fitness` collects a cumulative fitness score from a local fitness function for the flag assignments during loop execution.

Before the formal presentation of the algorithm, the transformation is illustrated to provide some initial intuition. To begin with, Figure 5.2(a) shows an untransformed program, which contains a single flag variable. In addition to serving as an illustration of the transformation, this program will be used in the empirical study because it denotes the worst possible case for structured code: as the size of the array `a` increases, the difficultly of the search problem increases. Metaphorically speaking, the needle (all array entries equal to zero) is sought in an increasingly larger haystack.

For illustration, suppose the goal is to execute the branch at node 6 in Figure 5.2(a). To realize this goal requires finding array values that avoid traversing the true branch of node 3 because if an input causes the program to pass through node 4, the target branch will be missed. The program in Figure 5.2(a) produces the landscape shown at the right of Figure 5.1. Transforming this program to count the number of times the predicate at node 3 is `false`, produces the landscape shown at the middle of Figure 5.1. The transformed program in shown in Figure 5.2(b). In essence, the counting drives the search away from executing node 4 because `fitness` receives a value closer to `counter` the more times node 4 is missed.

However, this *coarsely* transformed version does not provide the search with any guidance on finding inputs that make a particular array element zero. It only favours such inputs once found (thus the stair–step landscape in the middle of Figure 5.1). The *fine–grained* transformed version, shown in Figure 5.2(c) calls a *local* fitness function in the true branch of node 3 that helps guide the search towards individual array values being zero. In this case, the local fitness measures how close the input was at this point to avoiding node 4.

Node Id	Example
	`void f(char a[SIZE]){`
(1)	`int i, flag = 1;`
(2)	`for(i=0;i<SIZE;i++){`
(3)	`if(a[i]!=0)`
(4)	`flag=0;`
	`}`
(5)	`if(flag)`
(6)	`/*target*/`
	`}`

(a) No transformation

```
void f(char a[SIZE]){
   int i, flag = 1;
   double fitness, counter = 0.0;
   fitness = (-1) + (flag != 0);
   for(i=0;i<SIZE;i++){
      if (a[i] != 0){
         counter++;
         flag = 0;
      }else{
         fitness++;
         counter++;
      }
   }
   if(fitness == counter)
      /*target*/
}
```

(b) Coarse–grained transformation

```
void f(char a[SIZE]){
  int i, flag = 1;
  double fitness, counter, f;
  char __cil_tmp1, __cil_tmp2;
  counter = 0.0;
  fitness = (-1) + (flag != 0);
  for(i=0;i<SIZE;i++){
   if (a[i] != 0){
    __cil_tmp1 = a[i];
    __cil_tmp2 = 0;
    counter++;
    flag = 0;
    f =
    local(__cil_tmp1, "!=",
          __cil_tmp2);
    fitness += normalize(f);
   } else {
    counter++;
    fitness++;
   }
  }
  if(fitness == counter)
   /*target*/
}
```

(c) Fine–grained transformation

```
double normalize(double dist){
   return 1 - pow(1.001, -dist);
}

double
local(char arg1, char* op,){
      char arg2){
  double dist;
  if(strcmp(op, "!=") == 0){
     dist = abs(arg1 - arg2);
     if (dist == 0)
        return 0;
     else
        return (dist + 1);
  }
  else if(strcmp(op, "==") == 0){
     ...
  }
}
```

(d) Local fitness function

Figure 5.2: An example program before and after applying the coarse and fine–grain transformations. The figures also shows part of the function for computing local fitness.

Local fitness is computed by *negating* the predicate condition at node 3 and calculating a distance d for the negated predicate, based on a set of rules described by Bottaci (Bot02). In the example, d is equal to the ith value of `a`, indicating how close `a[i]` was to being 0 and thus traversing the false (desired) branch of node 3. Figure 5.2(d) presents a portion of the local fitness function used in the case of the example function. This portion is for the arithmetic comparison operator '`!=`'.

The formal transformation algorithm is presented in Figure 5.3. It assumes that `flag` is initially assigned `true` and might subsequently be assigned `false`. Clearly there is a complementary version of the algorithm which can be applied when the initial assignment to `flag` is `false`.

The rest of this section explains the algorithm's steps in detail. Step 1 ensures that all assignments to the variable `flag` are of the form `flag = true` or `flag = false`. This is achieved by replacing any assignment of the form `flag = C` for some boolean expression `C` with `if(C) then flag = true else flag = false`. Step 2 adds an empty `else` block to all `if` statements as a place holder for later code insertions. Steps 3 and 4 simply insert the fitness accumulation variable, `fitness`, and the assignment counter, `counter`, both initialized to 0 prior to the start of the loop.

Step 5 introduces the update of the fitness accumulation variable, `fitness`, and the loop counter, `counter`. It has three cases: the first, Case 5.1, checks for the special situation when the loop degenerates into a simple assignment. In Cases 5.2 and 5.3 the value added to `fitness` depends upon the value assigned to `flag` along the associated path. If `flag` is assigned `true` (Case 5.2) then, in essence, assignments in previous loop iterations are irrelevant. To account for this, `fitness` is assigned the current value of `counter` (after it has been incremented). This assignment overwrites any previously accumulated fitness.

Case 5.3 addresses an 'undesired' assignment to `flag`. In this case `flag` is assigned `false`. The CFG is used to identify the set of critical branching nodes for the flag assignment in Step 5.3.1. Critical branching nodes are those decision nodes in a CFG where the flow of control may traverse a branch which is part of a path that can never lead to the flag assignment. Note that the transformation ignores those critical branching nodes, which are also critical for the loop statement itself, as well as branching nodes which denote a loop exit condition. Step 5.3.2 iterates over all critical branching nodes and checks if they contain a branch CFG edge which is not post–dominated by

Step 1 Convert all flag assignments to assignments of constants by replacing `flag = C` with `if(C) then flag = true else flag = false` for some (side effect free) boolean expression `C`.

Step 2 Convert any `if - then` statements that contain a (nested) assignment of `flag` into `if - then - else` statements. The added empty branch is filled by Case 5.3 of Step 5 with 'bookkeeping' code.

Step 3 Add variable `counter = 0` as an initialization prior to the loop.

Step 4 Add an assignment `fitness = 0` as an initialization prior to the loop.

Step 5 There are three cases for assignments to `flag` based on the paths through the loop body.

Case 5.1: If all leaves of the AST contain the assignment `flag = false` (*i.e.* entering the loop means certain falseness), then the entire loop is treated as `flag = !C` assuming the original loop is `while(C)`. Otherwise, do the following for each leaf in the loop's AST that assigns to `flag`.

Case 5.2: `flag` is assigned `true`. Increment `counter` and assign value of `counter` to `fitness` immediately after the assignment to `flag`.

Case 5.3: `flag` is assigned `false`.

Step 5.3.1 Create a set, s_f, containing the critical branching nodes with respect to the flag assignment, and a set s_l containing the critical branching nodes with respect to the loop statement. Let π be the set difference between s_f and s_l, with all loop exit conditions removed from π.

Step 5.3.2 For every critical branching node in π, insert an increment for both `counter` and `fitness` as the first instructions in the `then` or `else` branches of the node that leads away from the flag assignment (*i.e.* the target of the branch CFG edge is not post–dominated by the flag assignment), if and only if, the target of the branch CFG edge is not post–dominated by another node in π, or by another assignment to `flag`. Do not add increments for `counter` and `fitness` otherwise.

Step 5.3.3 Collect the set of conditions s_c in π at which the assignment of `false` to `flag` can be avoided, *i.e.* the conditions of those nodes in π that contain a branch CFG edge whose target is post–dominated by the flag assignment. Step 5.3.1 ensures that such a condition exists.

Step 5.3.4 For each condition c in s_c do the following.

Step 5.3.4.1 Save the values of the variables used in c in well typed, local, temporary variables for later use (local with respect to the function body, not the enclosing block).

Step 5.3.4.2 Insert the call `f = local(...)` as the first instruction in the `then` or `else` branch of the node containing c that leads towards the flag assignment (*i.e.* the target of the branch CFG edge is post–dominated by the flag assignment). The function `local` is the standard local fitness function, and the temporary variables, alongside the binary operator used in c form the arguments of the function call `local`. As detailed in Section 5.4, the CIL infrastructure ensures c does not contain any logical operators.

Step 5.3.4.3 Normalize `f` to a value between 0 and 1.

Step 5.3.4.4 Add `f` to the existing value of `fitness` immediately after the flag assignment.

Step 5.3.4.5 Add an increment for `counter` immediately after the update to `fitness` (in Step 5.3.4.4).

Step 6 Replace `if(flag)` with `if(fitness==counter)`.

Figure 5.3: The transformation algorithm. Suppose that `flag` is assigned `true` outside the loop and that this is to be maintained.

either the flag assignment or any other critical branching node for the flag assignment. For each critical branching node which satisfies this requirement, Step 5.3.2 adds an increment of 1 to both `counter` and `fitness` as the first instructions to the branch that is not part of the path leading to the flag assignment. This also addresses the case when `flag` remains unassigned during a path through the loop.

Next, Step 5.3.3 collects the conditions of those branching nodes, which contain a branch CFG edge whose target is post–dominated by the flag assignment. For each of those conditions, Step 5.3.4 implements the more fine–grained approach producing a landscape more like that shown in the left of Figure 5.1. Smoothing of the fitness landscape improves the search. Here, if no changes to `fitness` were made, the resulting fitness landscape degenerates to the coarse–grained landscape shown in the middle of Figure 5.1. Instead Step 5.3.4 implements the transformation shown in Figure 5.2(c).

Steps 5.3.4.1 and 5.3.4.2 add the necessary instrumentation to compute a fitness increment for the path taken by an input. The result of the fitness computation is saved in a local variable, whose value is normalized in Step 5.3.4.3.

The key observation behind Steps 5.3.4.1 – 5.3.4.3 is that an assignment of `false` to `flag` occurs because a 'wrong decision' was taken earlier in the execution of the program. The algorithm therefore backtracks to this earlier point. That is, it finds a point at which a different decision (the decision c of Step 5.3.4) could avoid the assignment of `false` to `flag`. The value calculated (in Step 5.3.4.2) for the fitness increment in this case is based upon the standard approach to local fitness calculation in evolutionary testing (WBS01).

Finally, Step 5.3.4.4 adds the fitness increment to `fitness` immediately after the flag assignment, while Step 5.3.4.5 increments `counter`.

Step 6 replaces the use of `flag` with `fitness==counter`. Observe that the value of `fitness` can only equal the value of `counter` in two cases: either the last assignment to `flag` in the loop was the value `true` and there has been no subsequent assignment to `flag`, or the variable `flag` has not been assigned in the loop (so its value remains `true`). In either case, the original program would have executed the true branch of the predicate outside the loop which uses `flag`. In all other cases, `flag` would have been `false` in the original program. For these cases, the value of `fitness` will be some value less than that of `counter`. How close together their values are is determined by how close the loop comes to terminating with `flag` holding the desired value `true`.

It is important to note that the transformed program need not be semantically equivalent to the original. It is a new program constructed simply to mimic the behaviour of the original at the target branch. It does so in a way that ensures a more attractive fitness landscape. The standard search algorithm (with no modification) can be applied to the transformed program with the goal of finding test data to execute the branch controlled by the newly inserted predicate `fitness==counter`.

Finally, if `flag` is assigned in several loops, nested one within the other, then the algorithm is applied to the inner–most loop first in order to obtain a fitness value for the inner–most loop. This value can then be used as a partial result for the fitness of a single iteration of the enclosing loop. In this manner, the algorithm is applied to each enclosing loop, to accumulate a total fitness value.

5.4 Implementation

The algorithm has been implemented in a tool which is based on the CIL (NMRW02) infrastructure for C program analysis and transformation. CIL provides a number of predefined program analysis modules, such as control and data flow analysis. It also offers an extensive API (in Ocaml) to traverse the AST of the parsed source code. The tool itself is provided as an Ocaml module and can be run on any platform that has the Ocaml runtime installed.

5.4.1 Definition of Loop–Assigned Flag

For the purpose of the tool, a flag f is considered to be loop–assigned if, and only if it satisfies the following properties:

1. The definition f_{def} of f is a descendant of the loop statement l_s (*i.e.* a `while` or `for` construct) in the AST.

2. There exists a definition free path for f from f_{def} to f_{use}, where f_{use} is a predicate use of f.

3. f_{use} is not a descendant of l_s in the AST. If it is, it must also be a descendant of another loop statement $l_{s'}$ in the AST.

Flags assigned within loops that arise in a CFG as part of unstructured code (*e.g.* via the use of `goto` statements) are not considered to be loop–assigned by the tool, even though the algorithm proposed in Figure 5.3, in principle, does not necessitate this restriction. As a consequence, the tool might consider more flags to be loop–assigned than strictly necessary, while at the same time leaving loop–assigned flags that arise from unstructured code untransformed.

In general the C language does not contain a dedicated boolean data type[1], so the question remains how to identify flag variables. Generally speaking, since the aim of the testability transformation is to transform spikes in a fitness landscape, the transformation algorithm does not need to identify flags in a semantically correct way. The transformation can thus be applied to any variable whose use creates a spike in the fitness landscape. A syntactic check on predicates often suffices to identify such potential 'problem uses'.

Below are two examples of source code constructs that often cause a spike in the fitness landscape:

```
int foo(...){
  ...
  if(C)
    flag = 0;
  ...
  if(flag){//target}
  ...
}
```

```
int foo(...){
  ...
  if(C)
    buffer = malloc(...);
  ...
  if(buffer){//target}
  ...
}
```

Notice even though `buffer` is not a flag, the fitness landscape for the `//target` branch in the right column exhibits the same features as the flag controlled branch in the left column.

For the implementation described in this paper, a variable is considered to be a flag if it is of integral data type, and is used inside a predicate in one of these ways:

1. `if (variable)`
2. `if (!variable)`
3. `if (variable == constant)`
4. `if (variable != constant)`

[1] C99 defines a `_Bool` type which can take on the values zero and one.

5.4.2 Flag Removal

Before applying the transformation algorithm, the parsed source code needs to be simplified. To this end a number of CIL options, as well as some custom preprocessing transformations are used.

As mentioned in Section 3.2.3, CIL transforms compound predicates into equivalent `if - then - else` constructs. Besides the transformation of compound predicates, the tool requires the following code transformations prior to applying the flag transformation:

Simplify: This CIL option (`--dosimplify`) transforms the source code into simpler three–address code.

Simple Memory Operations: This option (`--dosimpleMem`) uses well–typed temporary variables in order to ensure each CIL lvalue involves at most one memory reference.

Prepare Control Flow Graph: This option (`--domakeCFG`) converts all `break`, `switch`, `default` and `continue` statements and labels into equivalent `if` and `goto` constructs.

Uninline: This custom step converts all in–line predicates (without logical operators) into equivalent `if - then - else` statements. Further, this module also implements Step 1 from Figure 5.3.

The implementation of Case 5.3.1 from Figure 5.3, requires control dependence information for each statement. CIL provides a module to compute a function's CFG by identifying each statement's control predecessor and successor, as well as a module to compute the immediate dominator information for a statement. The tool combines these two modules by first inverting the edges of the CFG (adding a unique exit node when necessary), and then computing the immediate dominator information for the inverted CFG. This is equivalent to computing the post domination information for each statement. Based on the post dominator tree, the control dependence information is computed for each statement in the AST.

Next, flags are collected by iterating over the CIL AST, performing a syntactic check on `if` statements. The preprocessing steps ensure that all predicates appear in the form of `if` statements. When a predicate matches the flag pattern described above,

information about the predicate and its parent statement (*i.e.* the `if` statement) are stored in a hash table.

For each entry in the hash table, the tool uses the CIL reaching definitions module to collect the set of definition statements for a flag reaching the predicate use of the flag. For each of these definitions, the tool checks whether they occur within a loop, and further that the flag use is not contained in the same loop. This is achieved by traversing the CIL AST. Loop–assigned flags are labeled as such.

For each loop based flag assignment, the control dependence information of the containing statement is used to derive a local fitness function. An example is given in Figure 5.2(d). All flag variables of a given type share the same local fitness function. The necessary type information can easily be extracted via a call to CIL's `typeof` function, which returns the data type of its argument. Finally, the statement containing the predicate use of the flag is transformed as described in Step 6 of Figure 5.3.

The tool can be run in two modes. By default, the local fitness function is used to 'punish' an assignment to flag, as illustrated in Figure 5.2(c). However, sometimes a flag assignment may be desired, and thus the transformation can be used to guide the test data generation process towards the statement containing the flag assignment. In this mode, `fitness` is incremented by the local distance function (not inverting its second parameter) in the branches avoiding the flag assignment.

5.4.3 Runtime

For the transformation to be applicable in practice, the tool should perform at a reasonable speed. The tool was therefore run on each of the test subjects used in Section 5.5, and timing information was recorded via the GNU `time` utility. The time measurements were collected on a HP Compaq 6715b laptop running Ubuntu version 9.04. For each test subject, the runtime of the tool was recorded five times to allow for slight variations in the measurements reported by `time`. The data, averaged over the five runs, is shown in Table 5.1. The tool did not require more than 2 seconds to apply the transformation to any of the test subjects.

5.4.4 Limitations

Currently the tool only implements an intraprocedural transformation for loop–assigned flags. As a result, global flags are ignored by the tool, as are flags passed by reference.

Table 5.1: Runtime of the transformation (in seconds) for the test subjects as reported by the `time` utility. The measurements are averaged over five runs. The column *real* refers to the wall clock time, *user* refers to the time used by the tool itself and any library subroutines called, while *sys* indicates the time used by system calls invoked by the tool.

Test Subject	real	user	sys
synthetic examples	0.1252	0.0440	0.0568
EPWIC	0.3092	0.1888	0.0896
bibclean	0.1752	0.0816	0.0632
ijpeg	1.8326	1.7232	0.0584
time	0.1654	0.0760	0.0648
plot2d	1.7412	1.6128	0.0752
tmnc	0.1738	0.0920	0.0544
handle_new_jobs	0.1548	0.0664	0.0632
netflow	0.1454	0.0568	0.0648
moveBiggestInFront	0.1306	0.0648	0.0520
update_shps	0.1664	0.0816	0.0608

Furthermore, the tool does not include any alias analysis and, as a consequence, does not handle intraprocedural or interprocedural aliasing of flag variables. The tool further distinguishes between function assigned flags and other flags. Function assigned flags are variables whose value depends on the return value of a function call. These types of flags can be handled by a different testability transformation (WBW07). Other kinds of flags include the loop–assigned flags addressed in this paper, and simply assigned flags addressed by previous work (AB06; HHH^{+}04; BS03; Bot02). For function assigned, simple and nested flags, the tool implements the transformation proposed by Wappler *et al.* (WBW07).

Both the algorithm presented in Figure 5.3 and the tool are incomplete in the presence of unstructured flow of control within the body of loops. Liu *et al.* (LLLW05) present a synthetic example which illustrates this incompleteness. In practice this limitation was only observed in 2 out of 17 functions examined during the empirical study in Section 5.5. Nevertheless this issue will be resolved in future versions of the tool.

5.5 Empirical Algorithm Evaluation

This section presents two empirical evaluations of the impact of the transformation algorithm. It first reports on the application of the transformation to the synthetic 'needle in the haystack' example from Figure 5.2(a). After this, it considers the application of the transformation to a collection of flags extracted from several production systems.

The synthetic benchmark program was chosen for experimentation because it potentially denotes the worst possible case for the search. To investigate how the transformation affects the performance of the test data generation technique with increasing problem difficulty, twenty versions of this program were experimented with. In each successive version, the array size was increased, from an initial size of 1, through to a maximum size of 40. As the size of the array increases, the difficultly of the search problem increases; metaphorically speaking, the needle is sought in an increasingly larger haystack. This single value must be found in a search–space, the size of which is governed by the size of the array, `a`. That is, the test data generation needs to find a single value (all array elements set to zero) in order to execute the branch marked `/* target */`.

The evaluation of the synthetic benchmark was done using an evolutionary testing system from Daimler (BSS02; WBS01) and AUSTIN. The results for the Daimler system had already been collected in the work by Baresel *et al.* (BBHK04) and are included in this thesis to facilitate comparisons with AUSTIN. In the work of Baresel *et al.* (BBHK04) the Daimler system had been applied to the syntethic benchmarks using no transformation, the coarse–grained transformation and finally the fine–grained transformation. Their results showed that the fine–grained transformation consistently outperforms the coarse–grained transformation. Hence, AUSTIN was only evaluated on the (standard) no transformation and fine–grained transformation approaches.

5.5.1 Synthetic Benchmarks

The analysis of the synthetic benchmark begins by discussing the results from applying the Daimler evolutionary testing system to the program without any transformation and after the fine–grained transformation (for results on the coarse–grained transformation the reader is referred to reference (BBHK04)).

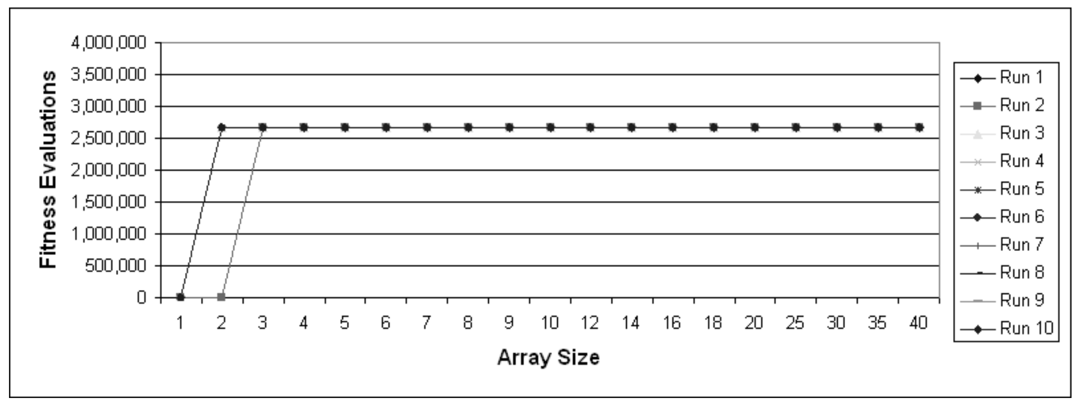
Figure 5.4 shows the results, with the 'no transformation' case shown at the top. As one would expect, the evolutionary search in essence deteriorates to a random search for the no transformation approach. The Daimler system only manages to find test data in two situations. The first is with an array size of one, where the search has a 1 in 256 chance of finding the needle in the haystack. Given this probability, the evolutionary search manages to find this needle in all of the 10 runs. The second instance where the evolutionary search successfully finds test data is with a two element array. The chance of randomly finding the required test data are reduced to 1 in $65,536$, and the Daimler system only manages to find the test data in one of 10 runs. Once the size of the array is increased to more than two elements, the evolutionary search consistently fails to find the required test data.

As the bottom graph of Figure 5.4 shows, the fine–grained approach on the other hand, enables the Daimler system to consistently find test data for all instances of the benchmark program. This provides evidence that the transformation is able to offer the search guidance towards the global optimum despite increasing problem complexity.

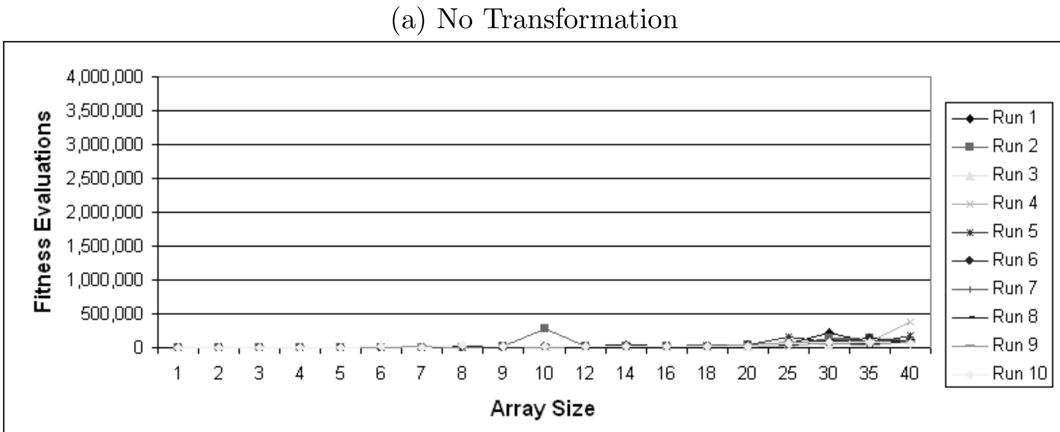
Next, AUSTIN was applied to the synthetic benchmark yielding similar results. The results using no transformation and the fine–grained transformations are depicted in Figure 5.6. As with the Daimler evolutionary testing system, AUSTIN fails to find any test data to cover the branch for array sizes greater than three when working with the untransformed program. In one run the search manages to cover the branch at an array size of three, in nine runs with an array size of two, and in all runs when the array contains only one element. For AUSTIN to cover the branch using the untransformed program, the search needs to either randomly find the solution, or randomly choose a starting point which is exactly one neighbourhood move away from the solution (*e.g.* `a = {-1,0,0}`, `a = {1,0,0}`, *etc.*).

For the fine–grained transformed program, AUSTIN requires far fewer fitness evaluations than the Daimler evolutionary testing system to cover the target branch. This is visually evident when comparing the y–axis scales of the chart in Figure 5.5, which shows the Daimler evolutionary testing system results, with that of the chart in Figure 5.6b, which shows the AUSTIN results. This difference is consistent with the findings of Harman and McMinn (HM09) and the results from Chapter 3.

Baresel *et al.* (BBHK04) reported a high standard deviation for the Daimler system for the coarse–grained transformation and a spike in standard deviation for an array



(a) No Transformation



(b) Fine–Grained Transformation

Figure 5.4: Results over ten runs of the evolutionary search for each of the two transformation approaches. The graphs have been taken from the paper by Baresel *et al.* (BBHK04).

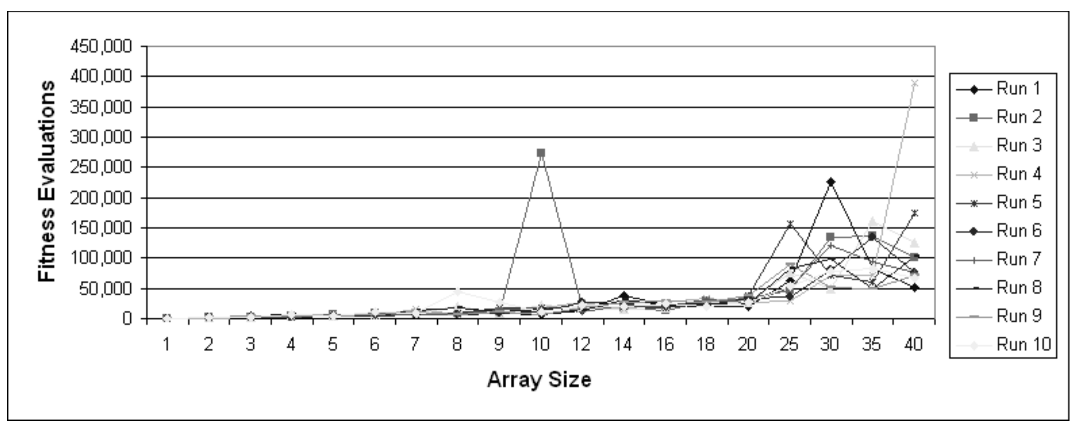
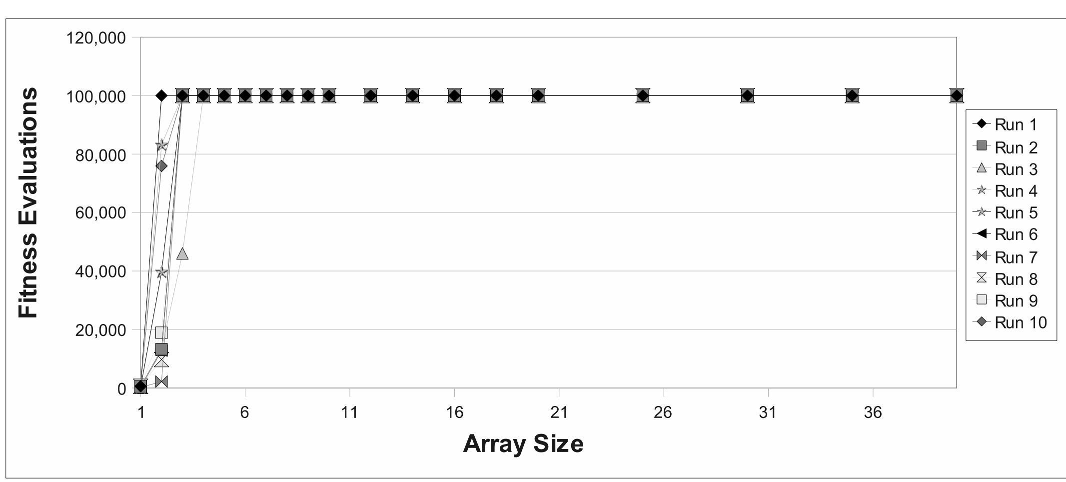


Figure 5.5: Results over ten runs of the evolutionary search for the fine–grained transformation approach close–up. The graph has been taken from the paper by Baresel *et al.* (BBHK04).

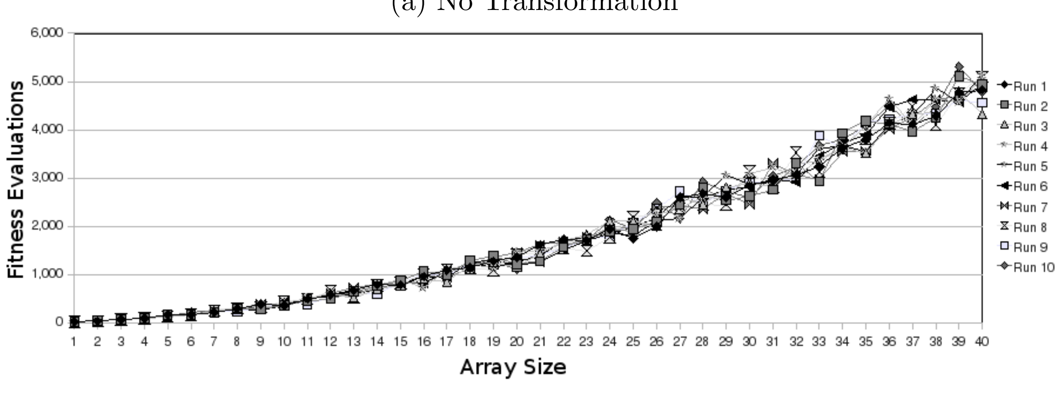
size of two elements in case of the 'no transformation' approach. This spike can be explained by the search randomly finding the test data for one of the runs. The pattern for AUSTIN is similar: the average for the 'no transformation' technique is almost uniformly the worst–case, while its standard deviation is zero, in all but the cases for array sizes 1, 2 and 3 (where some random chances led to a successful search). The high standard deviation for size 2 is again evidence that the one solution was a random occurrence.

The qualitative assessment presented in Figure 5.4 and Figure 5.6 clearly suggests that the fine–grained approach is better than the 'no transformation' approach. The 'trend' of the fine–grained transformation outperforming the 'no transformation' approach manifests itself as the size of the problem increases. It is also the case that as the difficulty of the problem increases (*i.e.* as the size of the array increases), the test data generation process will get harder for both, the fine–grained and 'no transformation' approaches. This can be seen in Figure 5.6b and Figure 5.5. Despite this, the fine–grained approach will always be able to offer the search guidance, whereas the 'no transformation' approach remains a random search for the 'needle' in an increasingly larger 'haystack'.

To complement this qualitative assessment quantitatively, an assessment using the Mann–Whitney test, a non–parametric test for statistical significance in the differences between two data sets, was performed. Because the test is non–parametric, the data



(a) No Transformation



(b) Fine–Grained Transformation

Figure 5.6: Results over ten runs of the alternating variable method for the 'no transformation' and fine–grained transformation approaches.

is not required to be normally distributed for the test to be applicable. The test reports, among other things, a p–value. The p–value for the test that compares the 'no transformation' results with the 'fine–grained transformation' results, returning a p–value of less than 0.0001, indicating that the difference is 'statistically significant at the 99% confidence level'.

The results obtained from the synthetic benchmarks clearly show that the test data generation process on the transformed (fine–grained) version outperforms the test data generation process on the untransformed version.

5.5.2 Open Source and Daimler Programs

In addition to the study of the synthetic program, which represents a problem that is synthetically generated to be hard for search–based approaches to solve, the seventeen C functions shown in Table 5.2 were used to evaluate the impact of the testability transformation of Figure 5.3. These functions were extracted from a combination of ten open source and Daimler programs. Each of the seventeen is first described in some detail to provide an indication of the flag usage in each program. This is followed by a discussion of the empirical results obtained using these functions.

`EPWIC` is an image compression utility. The selected function `run_length_encode_zeros` loops through a stream of data (encoded as a C string) and counts the number of consecutive occurrences of 0. If a zero is found, a counter is incremented and the flag `found_zero` is set. The flag is initialized to 1 and the test problem is to find inputs which avoid setting the flag to 0.

`bibclean` is a program used to check the syntax of BibTeX files and pretty print them. Two functions were selected. The first, `check_ISBN`, loops through an input string to check whether it represents a valid ISBN number. It contains the flag `new_ISBN`, which is initialized to 1 and set to 0 at the start of every loop iteration whenever its value is 1. The flag is reset to 1 in the body of the loop only when a given number of parsed characters in the range $0 - 9$, including 'x' and 'X', represent an invalid ISBN number, or, the string does not represent a valid ISBN number at all, but contains more than 10 valid ISBN characters. The test data generation challenge is to discover a string with more than 10 valid ISBN characters which do not represent a valid ISBN number. The search has to

Table 5.2: Test subjects for the evaluation of the transformation algorithm.

Test subject	Function under test
EPWIC	run_length_encode_zeros
bibclean	check_ISBN check_ISSN
ijpeg	emit_dqt format_message next_marker pbm_writepbmrowraw write_frame_header
time	getargs
plot2d	CPLOT_DetermineSeriesStatistics
tmnc	rule_I_intercept
handle_new_jobs	handle_new_jobs
netflow	netflow
moveBiggestInFront	moveBiggestInFront
update_shps	update_shps

navigate through the entire input domain of the function, which is approximately 10^{120}. The second function, `check_ISSN` works exactly as `check_ISBN` except that ISSN instead of ISBN numbers are checked for.

`ijpeg` implements an image compression and decompression algorithm, of which five functions were tested. The first, `emit_dqt`, contains the flag `prec`. An array of 64 unsigned integers is iterated over. If an element exceeds the value 255, the flag is set to `true`. The target branch is dependent on the flag being set to `true`.

The second function, `format_message`, formats a message string for the most recent JPEG error or message. It contains the flag `isstring`, which is used to check if the format string contains the '%s' format parameter. The test data generation problem is to find an input string which contains the character sequence '%s'.

The third function, `next_marker`, contains a loop–assigned flag `c`, which is part of the termination criterion for a `do {} while()` loop. An input buffer is traversed and the current character assigned to `c`. The loop terminates if `c` is not equal

to 255, thus the challenge is to discover that an array of inputs is required that contains the value 255 at least once.

The fourth function, `pbm_writepbmrowraw`, contains the local variable `bitshift` of type `int` which is initialized to 7. The inputs to the function are a file pointer, a pointer to the start of a row in a matrix of unsigned character types, and the number of columns in the row. A loop goes through each column and checks its entry for a non–zero value. Whenever a non–zero character is encountered, the value of `bitshift` is decremented by one. When `bitshift` takes on the value -1 it is reset to 7. After the body of the loop, the function checks for `bitshift` not being equal to 7. In this case the hard to cover target branch is the false branch.

The final function, `write_frame_header`, contains the loop–assigned flag `is_baseline`, which is initialized to 1 and assigned 0 in the body of the loop. The target branch depends on the flag retaining its initial value. To complicate matters, the flag may be initialized to 0 before the start of the loop if two properties of the function's input domain are true. This assignment is not part of the transformation *per se* (apart from ensuring that the state of the flag is correctly represented by the helper variables regardless of the path taken to reach the start of loop), thus it remains an additional goal of the search algorithm to find inputs which avoid initializing the flag to 0.

`time` is a GNU command line utility which takes as input another process (a program) with its corresponding arguments and returns information about the resources used by the process (*e.g.*, the wall–clock and CPU time used). The function `getargs` contains three loop–assigned flags, `outfile`, `append` and `verbose`. The function parses the command line arguments and sets the option flags accordingly. The challenge during the test data generation process is to find input parameters encoded as strings, that are valid options, setting these flags.

`plot2d` is a small program that produces scatter plots directly to a compressed image file. The core of the program is written in ANSI C. However the entire application includes C++ code. Only the C part of the program was considered during testing. The function `CPLOT_DetermineSeriesStatistics` contains the loop–assigned flag `computeStats`, which is initialized with 1 and only ever assigned 1

in the body of the loop. The branch dependent on the `false` outcome of the flag is therefore infeasible and the true branch trivially covered.

`tmnc` is a C implementation of the TMN protocol. The function `rule_I_intercept` loops through an array of sessions (containing, amongst other things, information about the initiator and responder), validating a session object. If the session is valid, a flag is set.

`handle_new_jobs` is a job scheduler responsible for management of a set of jobs stored in an array. Each job has a status and priority, as well as additional data used during job execution. This code is part of the Daimler C++ testing system itself: it facilitates parallel execution of test processes. The input space is the job array (the 'data' entries are unimportant for coverage). The test problem is to find the right input data for the flag, `check_work`, tested in the last condition of the function. In order to execute the true branch of this conditional, the assignment `check_work = 1;` in the `for` loop must be avoided in every iteration.

`netflow` is part of an ACM algorithm for performing net flow optimization. The function has many input parameters configuring the net to be optimized, for example connected nodes and connection capacity. The two parameters of the function are `low` and `high`. The netflow function begins with some plausibility checks on the input parameters. The flag variable `violation` is typical of a test for special conditions which cannot be handled by the regular algorithm. As an invalid input check, `violation` is set to `true` when `low` is set to a larger value than `high`.

`moveBiggestInFront` is part of a standard sorting algorithm. A `while` loop processes the elements of an array, checking whether the first element is the biggest. If no such value exists, this constitutes a special case with the result that the flag assignment is not executed in any iteration.

`update_shps` is a navigation system used by Daimler in vehicular control systems. The code has been modified to protect commercially sensitive information. However, these modifications do not affect the properties of the code with respect to flag variable use. The navigation system operates on a 'Shape Point Buffer' which

stores information from a digital street map. Streets are defined by shape points. The buffer contains map locations (points) near to the current location of the car. For testing, the input space is formed from the set of shape point buffer data stored in a global array and the position of the car supplied as the parameters of the function. The function uses a flag, `update_points`, to identify a situation where an update is required. The flag is assigned inside a loop traversing the shape point buffer. The flag becomes `true` if any shape point from the buffer is outside a certain area. The target branch is hard to execute because input situations rarely lead to `update_points` being assigned `false`. The search–space for the predicate `if(!update_points)` is precisely the worst–case flag landscape shown on the right side of Figure 5.1.

These seventeen functions capture the full range of difficulties for the search. At the easy end of the spectrum, test data for the flag use in the predicate from `plot2d` was always found in a single evaluation, both before and after transformation. Code inspection revealed that every path through the loop (and in fact the function) assigned `true` to the flag. Prior to the body of the loop, the flag is initialized to `true`. After the loop, the function contains a check for the value `true` of flag. Since the false branch of this check is clearly infeasible, it is not clear if this code was written anticipating some structural addition; perhaps it is a bug.

At the other end of the spectrum, the evaluation budget was exhausted in both the transformed and untransformed versions of three test subjects: `getargs_append`, `getargs_verbose`, and `next_marker`. The first of the three, `getargs_append`, comes from the command line argument processing of the program `time`. This example uncovered another limitation in the current AUSTIN and Daimler tool implementations. The tools do not properly handle C `static` variables, which are used to hold values across multiple calls to a function. In this case, `time` uses the function `getopt`, which records an index and look–ahead character in the `static` variables `optind` and `nextchar`. These two `static` variables effectively prevent the search from returning to a previous solution after exploring an inferior neighbour. This bug has now been fixed in AUSTIN.

The second function `getargs_verbose`, also from `time`, and the third `next_marker` from `ijpeg` both contain unstructured control flow. In this case an exit statement

within the loop, though the impact of statements like `return`, `break` and `continue` would be similar. In essence, these statements prevent the search from exploiting accumulated fitness information. In both cases, the search does not execute the transformed predicate (created by Step 6 of the algorithm shown in Figure 5.3).

Observe that none of the aforementioned issues denote flag problems. Rather, the application of search–based testing techniques to real–world programs has thrown up subsidiary issues and barriers to test data generation that have nothing to do with flags. It should be recognized that no 'perfect' solution to the test data generation problem yet exists; all techniques have language features that present difficulties. The purpose of this chapter is to demonstrate that the barrier to search–based testing raised by the presence of flag variables can be lowered by the testability transformation approach advocated in the chapter. However, there will remain further work required on these other issues of search–based testing.

The remaining thirteen functions fall in between these two extremes. In each case, the transformation improves the search for test data. The average success over all ten runs of each test subject is reported in Table 5.3. This table and Table 5.4 are arranged based on the number of fitness evaluations performed using the untransformed program. Overall the transformation led to a 28% increase in successful test–data generation.

Table 5.4 shows the number of fitness evaluations used by all ten runs for each program. This data is shown graphically in Figure 5.7, which uses a log scale on the y–axis. Overall, the transformation leads to a 45% improvement, reducing the total number of evaluations needed from $4,589$ to $2,522$, which represents a statistically significant reduction (student's t–test p–value $= 0.031$). It produces an improvement for all but four of the test subjects. In several cases the improvement is dramatic, for example, with the function `moveBiggestInFront` the total number of evaluations drops from $9,011$ to 10. Even in cases where the untransformed programs never exhausts its evaluation budget, there are large improvements, for example, with the function `format_message` there is a 91% decrease from $2,811$ to 245.

However, these results have to be treated with statistical caution. The average results of 10 runs for each flag use of each predicate in each function studied was taken. These average values form a sample from two paired populations: the 'with treatment' population and the 'without treatment population' for flag uses in predicates. In this

Table 5.3: Branch coverage results from empirical study of functions extracted from open source software.

Program	Function	Covered Branches untrans -formed	Covered Branches trans -formed
time	getargs_append	0	0
time	getargs_verbose	0	0
ijpeg	next_marker	0	0
ijpeg	write_frame_header	0	10
netflow	netflow	0	10
moveBiggestInFront	moveBiggestInFront	3	10
tmnc	rule_I_intercept	2	4
EPWIC	run_length_encode_zeros	10	10
ijpeg	format_message	10	10
bibclean	check_ISBN	10	10
bibclean	check_ISSN	10	10
ijpeg	emit_dqt	10	10
time	getargs_outfile	10	10
update_shps	update_shps	10	10
handle_new_jobs	handle_new_jobs	10	10
ijpeg	pbm_writepbmrowraw	10	10
plot2d	CPLOT_DetermineSeriesStatistics	10	10
	average	6.2	7.9
	percent improvement		28%

Table 5.4: Fitness function evaluations from empirical study of functions extracted from open source software.

Program	Function	Fitness Evaluations untrans -formed	trans -formed	savings	percent reduction
time	getargs_append	10,000	10,000	0	0%
time	getargs_verbose	10,000	10,000	0	0%
ijpeg	next_marker	10,000	10,000	0	0%
ijpeg	write_frame_header	10,000	412	9,588	96%
netflow	netflow	10,000	61	9,939	99%
moveBiggestInFront	moveBiggestInFront	9,011	10	9,001	100%
tmnc	rule_I_intercept	8,767	8,451	316	4%
EPWIC	run_length_encode_zeros	3,401	2,066	1,335	39%
ijpeg	format_message	2,811	245	2,566	91%
bibclean	check_ISBN	1,385	543	842	61%
bibclean	check_ISSN	843	664	179	21%
ijpeg	emit_dqt	835	145	690	83%
time	getargs_outfile	478	218	260	54%
update_shps	update_shps	271	45	226	83%
handle_new_jobs	handle_new_jobs	202	6	196	97%
ijpeg	pbm_writepbmrowraw	7	5	2	29%
plot2d	CPLOT	1	1	0	0%
	average	4,589	2,522		
	percent improvement		45%		

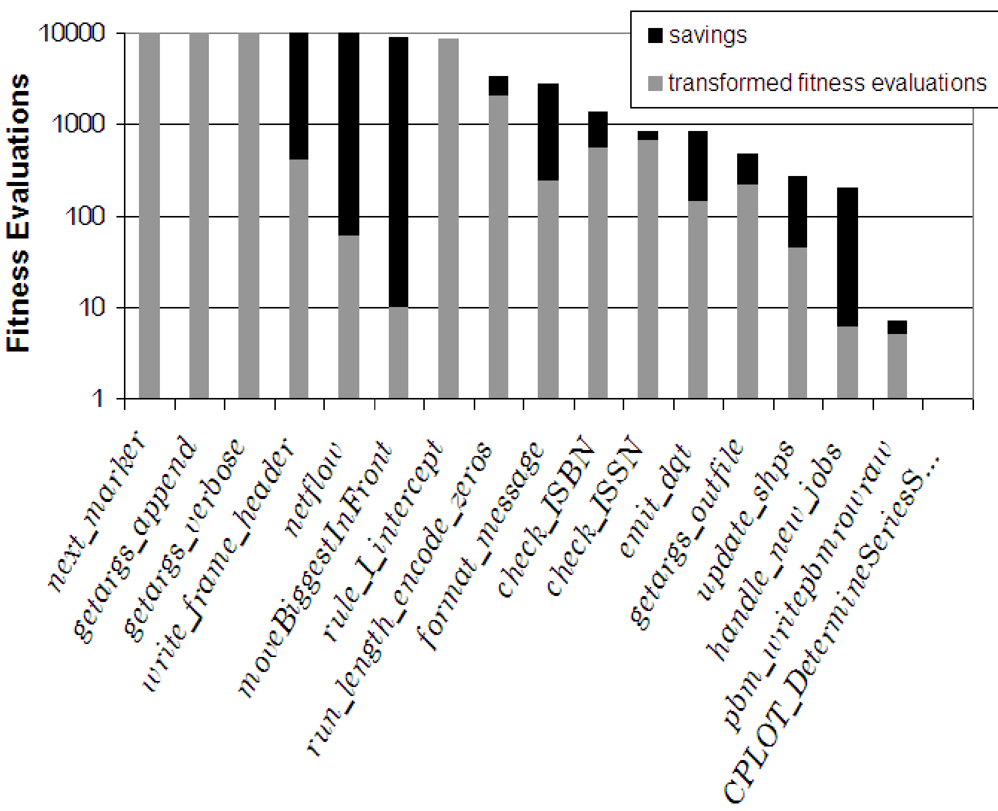


Figure 5.7: Chart of data from second empirical study.

case, 'with treatment' means with some form of transformation aimed to improve the test data generation process.

The samples involved in this test are not (and, indeed, cannot be) sampled in an entirely random and unbiased manner. They are samples from the space of all possible loop–assigned flag uses in all predicates in all functions in all C programs. There is even an issue here as to what constitutes the 'population': should it be, for example, all C programs possible, all those currently written, or all those in use in production systems? These questions bedevil any attempt to make reliable claims for statistical significance of results for studies involving samples of program code. The statistical tests are merely used to give a rough indication of the strength of the results for the programs studied, rather than to make claims about the behaviour of untried predicates in programs as yet unconsidered in the study.

5.6 Relevant Literature For the Loop–Assigned Flag Problem

The loop–assigned flag problem discussed in this chapter is relevant for all search–based test data generation methods, including the chaining approach (FK96; MH06). The chaining approach tries to identify sequences of nodes in a program's control flow graph which need to be executed in order to reach a specified target branch. Loop–assigned variables may lead to 'infinite' chains or result in loss of information because it is not known a priori how often a node inside a loop needs to be executed.

For a subset of a C–like language, Offutt *et al.* introduce a dynamic variant of constraint solving that performs dynamic domain reduction (OJP99). The algorithm can be applied to the flag problem. Firstly a path to the target is selected for execution, then the domains of each input variable are refined as execution follows this path. Domain refinement takes place at assignments and decision statements. For example, if the domain of `a` and `b` were both $1 \ldots 10$ before `if (a != b)`, then in the true branch of the `if` statement, `a` and `b` would be assigned the domains $1 \ldots 5$ and $6 \ldots 10$ respectively.

Loops are handled in a similar fashion by marking the loop predicates and dynamically reducing the input domain of the variables involved in loop constraints. However, the domain reduction requires knowing a priori a path to the target. Thus, for this dynamic domain reduction technique to cover the flag controlled branch in the program shown in Figure 5.2(a) requires that it first selects the path through the body of the loop which avoids each assignment to flag. This selection is essentially done by chance and the set of feasible paths is large, therefore this may take some time.

Flags often present the worst–case scenario to search–based test data generation techniques, particularly when only very few sub–paths will result in a flag taking on one of its two values. In contrast, the approach presented in this chapter is able to offer the search coarse and fine–grained guidance. This makes the approach more applicable to the flag problem in the presence of loop–assigned flags.

Search–based testing in the presence of flags has been studied by four authors (Bot02; BS03; HHH$^+$04; LLLW05). Bottaci (Bot02) aimed to establish a link between a flag use and the expression assigning a flag. This is done by storing the fitness whenever a flag assignment occurs so it can be used later on.

Baresel and Sthamer (BS03) use a similar approach to Bottaci. However, whereas Bottaci's approach is to store the values of fitness as the flag is assigned, Baresel and Sthamer use static data analysis to locate the assignments in the code, which have an influence on the flag condition at the point of use. Baresel and Sthamer report that the approach also works for enumeration types and give results from real–world examples, which show that the approach reduces test effort and increases test effectiveness.

Harman *et al.* ($\text{HHH}^{+}04$) illustrate how a testability transformation originating from an amorphous slicing technique can be used to transform flag–containing programs into flag–free equivalents. They achieve this by substituting a flag use with the condition leading to, as well as the definition of a flag, with the aid of temporary variables.

Liu *et al.* ($\text{LLW}^{+}05$) present an approach for unifying fitness function calculations for non–loop–assigned flags, and consider the problem of loop–assigned flags in the presence of `break` and `continue` statements (LLLW05).

Three of these approaches share a similar theme: they seek to connect the last assignment to the flag variable to the use of the flag at the point where it controls the branch of interest. In Bottaci's approach the connection is made through auxiliary instrumentation variables, in that of Baresel and Sthamer it is made through data flow analysis and, in the approach of Harman *et al.*, a literal connection is made by substitution in the source code.

The algorithm presented in Figure 5.3 and the approach by Liu *et al.* could be thought of as a combination of the approaches of Bottaci and Harman *et al.* They share the use of auxiliary 'instrumentation variables' with Bottaci's approach, but use these in a transformed version of the original program using transformations like the approach of Harman *et al.*

Alshraideh and Bottaci (AB06) proposed an approach to increase diversity of population–based test data generation approaches, for situations where the fitness function results in plateaux. This partly addresses issues relating to plateaux by increasing the chances that random mutations will move the population off the plateau. One of the underlying features of the flag problem is the way in which plateaux are present. In the case of hard–to–test flags, the landscape is formed of one very large plateau (with a tiny spike; the needle in the haystack) and, even with increased diversity, the search–based approach reduces to random search.

All of the evolutionary approaches can benefit from general improvements in genetic algorithms. For example, Godefroid and Khurshid consider a framework for exploring very large state spaces often seen in models for concurrent systems (GK04). They describe an experiment with an implementation based on the VeriSoft tool. They experiment with heuristics that improved the genetic algorithm's mutation operators and also show how Partial–order reduction can allow a greater search–space to be considered.

Finally, from a transformation standpoint, the algorithm presented here is interesting because it does not preserve functional equivalence. This is a departure from most previous work on program transformation, but it is not the first instance of non–traditional–meaning preserving transformation in the literature. Previous examples include Weiser's slicing (Wei79) and the 'evolution transforms' of Dershowitz and Manna (DM77) and Feather (Fea82). However, both slices and evolution transforms do preserve some projection of traditional meaning. The testability transformation introduced here does not; rather, it preserves an entirely new form of meaning, derived from the need to improve test data generation rather than the need to improve the program itself.

5.7 Conclusion

This chapter presented a testability transformation that handles the flag problem for evolutionary testing. Unlike previous approaches, the transformation introduced here can handle flags assigned in loops. Also, unlike previous transformation approaches (either to the flag problem or to other more traditional applications of transformation) the transformations introduced are not meaning preserving in the traditional sense; rather than preserving functional equivalence, all that is required is to preserve the adequacy of test data.

An implementation of the algorithm is discussed. The implementation is based on CIL, an infrastructure for C program analysis and transformations. The modular nature of both CIL and the tool allows the system to be extended in the future and incorporate different transformation algorithms, thus forming an effective transformation tool, geared toward improving evolutionary testing.

The behaviour of the algorithm is evaluated with two empirical studies that involve a synthetic program and functions taken from open source programs. The synthetic examples are used to illustrate how two variations of the algorithm perform for different levels of difficulty. The results show that the approach scales well to even very difficult search landscapes, for which test data is notoriously hard to find. The worst–case considered involves finding a single adequate test input from a search–space of size 2^{320}. Despite the size and difficulty of this search problem, the search–based testing approach, augmented with the transformation algorithm introduced here, consistently finds this value.

Chapter 6

Multi–Objective Test Data Generation

6.1 Introduction

In all previous work on search–based branch adequate test data generation, the problem has been formulated as a single–objective search problem: the sole objective is to cover the branch in question. While this is valuable, in many situations the tester may have additional goals they would like to achieve using the same test set. For example, the tester may also wish to find test cases that are more likely to be fault–revealing, or test cases that combine different non–subsuming coverage based criteria. The tester might also be concerned with test cases that exercise the usage of the stack or the heap, potentially revealing problems with the stack size or with memory leaks and heap allocation problems. There may also be additional domain–specific goals the tester would like to achieve, for instance, exercising the tables of a database in a certain way, or causing certain implementation states to be reached. In any such scenario in which the tester has additional goals over and above branch coverage, existing approaches represent an over–simplification of the problem in hand. A multi–objective optimization approach would be more realistic. The lack of any previous work on multi–objective test data generation therefore highlights an important gap in the literature. This chapter takes a first step towards filling this gap. It formulates the test data generation problem as a multi–objective optimization problem, presenting results from a study of a Pareto optimal approach and a weighted fitness approach.

In order to scope the problem, it was necessary to consider a suitable multi–objective scenario. As the previous discussion indicates, there are many choices. The scenario adopted is one in which the tester wishes to achieve branch coverage, while also constructing test cases that exercise the dynamic memory allocation of the program under test. This scenario would occur where, for example, the tester knows that memory is highly constrained or where the tester believes that there may be memory leaks or possible null pointer dereferences. The chapter presents results from five case studies of this multi–objective problem when applied to real code from the Software–artifact Infrastructure Repository (SaIR) and also to specially constructed examples that denote extreme cases where the two objectives are either in full agreement or total opposition. These extreme cases allow the approaches to be explored at the limits for which they might be expected to be applied. The primary contributions of this chapter are as follows:

1. The chapter introduces the first formulation of test data generation as a multi–objective problem. It describes the particular goal oriented nature of the coverage criterion, showing how it presents interesting algorithmic design challenges when combined with the non goal oriented memory consumption criterion.

2. The chapter presents results that confirm that multi–objective search algorithms can be used to address the problem, by applying the 'sanity check' that search–based approaches outperform a simple multi–objective random search.

3. The chapter presents results that suggest that a suitably constructed weighted multi–objective approach, though simplistic, can be effective for this problem in some cases. However, there is also evidence that a Pareto optimal approach can find better solutions in other cases, with respect to a certain objective. The chapter shows how a weighted and Pareto hybrid approach can be used to complement each other.

4. The chapter also presents results from the application of a Pareto optimal evolutionary algorithm, assessing the impact of the interdependencies that arise between the set of search problems denoted by the set of branches to be covered.

The rest of this chapter is organised as follows. Section 6.2 provides an overview of background information on multi–objective evolutionary algorithms. Section 6.3

describes two different approaches for attempting branch coverage while maximizing dynamic memory allocation. It also outlines the NSGA–II algorithm by Deb *et al.* (DAPM00) used in one approach. Sections 6.4 and 6.5 present the experimental setup and five case studies comparing both approaches when applied to synthetic and real–world programs. Insights gained during the studies are revealed in Section 6.6 and Section 6.7 concludes.

6.2 Background

Multi–objective evolutionary algorithms are algorithms designed for solving problems where no single optimal solution exists and a set of solutions is required instead. An example of a mutli–objective problem is the Knapsack problem (MT81), where weight has to be minimized and profit maximized. This is also a typical example where two objectives are in direct conflict with each other.

As stated, MOPs require a set of solutions known as a Pareto optimal set. Such a set contains only non–dominating solutions. The concept of domination is defined as:

> Individual X dominates Y if, and only if, X is better than Y in at least one objective, and no worse in all other objectives.

A Pareto front and Pareto optimal set can be defined as ((HBC$^+$04))

> **Pareto Optimal Set:** For a given MOP $\vec{f}(\vec{x})$, the Pareto Optimal Set ($P*$) is defined as: $P* := \{\vec{x} \in F \mid \neg\exists\vec{x'} \in F \; \vec{f}(\vec{x'}) \leq \vec{f}(\vec{x})\}$, where F is the decision variable space.

Genetic algorithms are naturally suited for MOPs because they maintain a number of individuals in every population, each of which may be better suited for one objective than another. Genetic algorithms can also optimize multiple objectives in parallel. A Pareto GA exploits this feature, giving it the ability to generate a Pareto optimal set in a single run.

A special type of single–objective GA, such as a weighted GA, might achieve the same results over a number of runs. However, for each run, the objectives have to be formulated as a set of constraints rather than optimization problems.

Most MOPs, such as the Knapsack problem mentioned above, have a large set of 'good compromise' solutions compared to the set of 'unpractical' solutions. For example, the only 'useless' solutions for the Knapsack problem are the zero weight or minimal profit solutions, *i.e.* the very extreme points on a Pareto front.

The opposite is true for the MOP addressed in this chapter. Branch coverage is attempted at a sub–goal level, with each goal corresponding to a branch in a program. It follows that branch coverage can only have a boolean outcome; either a test case traverses the branch or it fails to do so. The latter test cases are of no practical use, therefore branch coverage has to be achieved, even if this should mean failing to allocate any memory. Hence, the set of desirable solutions is very restricted; it may only contain one solution.

This observation would suggest that a weighted GA is better suited for this problem. However, as memory allocation may be optimizable for some branches (*e.g.* by increasing loop–iterations which allocate memory), a Pareto GA might also be adequate, especially since finding the right set of 'weights' can prove challenging and a different set of weights might be required for each branch.

In a weighted GA, each objective is given a coefficient, acting as a 'weight' for its fitness value. The fitness values for each objective are then combined into a single value, from which point onwards the multi–objective GA becomes identical to a single–objective GA. In order for a weighted GA to be effective, particularly when two objectives are in conflict, the objectives have to be ordered or prioritized in some way. This is in contrast to a MOEA, which treats all objectives as equally important.

For the problem considered in this chapter, one objective, maximizing memory allocation, is not clearly definable or quantifiable, because memory consumption may not have an obvious 'optimum'. Giving it too much weighting might inhibit branch coverage, because of the adverse effect on the overall fitness value (depending on the code/inputs). On the other hand, excessively reducing its weighting might render it insignificant. The aim of maximizing memory allocation would thus be reduced to a random search.

6.3 Implementation

The IGUANA tool (McM07) was adapted for the implementation of the two algorithms, which are based on a model described by Wegener *et al.* (WBS01). In order to measure memory allocation, the source code was instrumented with a global variable, used to count all bytes allocated during the execution of a function. The freeing of memory by deallocating memory pointed to by a pointer was not accounted for in this study because all the memory was allocated to global pointers which were released outside the scope of the function under test. The next section describes the operations used by the GAs and their configuration.

Two different types of selection operators were used for the implementation of the weighted and Pareto GA. For the Pareto approach an elitist selection and reinsertion strategy was chosen. Elitism ensures that the current best individual (or a set of best individuals in case of a multi–objective GA) is copied across into the next generation. The weighted GA uses stochastic universal sampling (Bak87) as a selection method, where the probability of an individual being selected is proportionate to its fitness value. This means ‘fitter’ individuals have more chance of being selected, but an ‘unfit’ individual may still be included, thereby partially maintaining diversity within the population in order to prevent a premature convergence at a sub–optimal solution.

Before individuals are selected for crossover, they must be ranked according to their fitness value within the population. The weighted-GA uses linear ranking (Whi89) with a selection pressure Z of 1.7 (in accordance with the Wegener model), where ordered individuals are assigned fitness values such that the best individual has a fitness of Z, the median individual a fitness of 1.0 and the worst individual a fitness of $2 - Z$, where $Z = [1.0, 2.0]$. These ranked values are then converted into proportionate fitness values before selection takes place. Ranked fitness values for the Pareto GA are calculated based on the Pareto–ranking method described in Section 6.3.1.

Discrete recombination (MSV93) was used to produce offspring and the mutation algorithm is based on the breeder genetic algorithm (MSV93). It defines a mutation probability of $1/len$, where len is the length of the input vector. Each of the breeding populations contains a different mutation step size p, ranging from 0.1 to 0.000001. A mutation range r_i is defined for each input parameter x_i by the product of p and the domain size of x_i, with $0 \leq i < len$. The ‘mutated’ value v_i of x_i can thus be computed

as $v_i = x_i \pm r_i * \delta$. Addition or subtraction is chosen with an equal probability. The value of δ is defined to be $\sum_{x=0}^{15} \alpha_x * 2^{-x}$, where each α_x is 1 with a probability of 1/16 else 0. If v_i is outside the allowed bounds of x_i, its value is set to either the minimum or maximum value for x_i.

A competition manager controls the number of individuals each population evolves. Every 20 generations, 10% of the individuals are randomly chosen for migration from one population to another. A migration manager ensures a population will only receive individuals from at most one other population. The competition manager also calculates a progress value for each population at the end of a generation. This progress value p is computed for a population at generation g as follows: $0.9 * p + 0.1 * rank$. $rank$ indicates the average fitness of a population and is obtained by linearly ranking the individuals within a population, as well as the populations amongst themselves. Again, a selection pressure of 1.7 was used for obtaining each rank value.

Every n number of generations, where n is configured via the competition manager, the populations are ranked according to their progress values. After this, a reallocation parameter is computed for each population, which controls how many individuals from the worst performing populations are transfered to the best performing ones. However, a population is not allowed to loose its last five individuals to prevent it from dying out. The transfer of individuals between populations is aimed at improving the overall performance of the GA. Thus, in effect, the best performing population is seeded every n number of generations and, as a result, will contribute most to the number of fitness evaluations.

6.3.1 Pareto Genetic Algorithm Implementation

The implementation adapted for this chapter is based on the NSGA–II algorithm described by Deb *et al.* (DAPM00). The main difference between a standard GA and a multi–objective GA is the way fitness values are computed and individuals ranked within a population. The NSGA–II algorithm creates a set of front lines, each front containing only non–dominating solutions. Within a front, individuals are rewarded for being 'spread out'. The algorithm also ensures that the lowest ranked individual of a front still has a better fitness value than the highest ranked individual of the next front.

The remainder of this section explains the ranking algorithm's steps in detail.

The first stage of the algorithm calculates two entities for each solution (DAPM00): 1) a count c for the number of individuals which dominate the current individual; 2) a set of individuals which are dominated by the current individual.

All individuals with a count of zero, *i.e.* those not dominated by any other individual, are grouped together to form the first front. The individuals from this front are then iterated and for each individual in their 'domination set', the count is reduced by one. Individuals that subsequently end up with a count of zero are again grouped together to form the next front. This process is repeated until all individuals are assigned to a front.

In order to encourage diversity within a front and prevent premature convergence, individuals are rewarded for lying either at extreme ends or less crowded regions of a front. This is done by assigning a 'distance' attribute d to each individual, which measures the distance of an individual to its closest neighbours. The value of d is defined to be $\sum_{x=0}^{n-1} \delta_x$ where n is the number of objectives and δ_x the combined distance of an individual from its closest neighbours with respect to the current objective.

Thus, if two individuals have the same non–domination count c, the individual with the greater 'distance' d ranks higher.

6.3.2 Weighted Genetic Algorithm Implementation

Unlike the Pareto GA, a weighted GA can only find a single best solution. This approach is commonly applied to multi–objective problems where it is possible to prioritize or order objectives in a meaningful way.

Branch coverage is a minimization task, where an ideal solution has an objective value of zero. In order to combine the objective value for the memory allocation with the distance measures, the inverse of the normalized number of bytes allocated was used. As this value can never reach zero, a cut–off point of 10^{-5} was chosen as the 'ideal' memory value and thus ideal overall fitness. If a branch fails to allocate any memory, a worst–case value of 1000 was used as the objective value for the memory allocation.

Thus, if a test case allocates memory, the formula used to obtain the objective value is: $1.001^{-b} * w_b^{-1} + 1.0 * d$, where b is the *raw* number of bytes allocated, w_b is the weight for the memory objective, with $0 < w_b \leq 10^5$, and d the distance measure composed of *branch distance* and *approach level*. The weight for d was left constant at 1.0.

6.4 Experimental Setup

Five case studies were carried out into the effectiveness of different search methods in generating branch adequate test data while maximizing dynamic memory allocation. The three searches considered are: a random search, Pareto optimal and, a weighted search. Two case studies are based on real–world C code and three on synthetic programs. The synthetic programs were chosen to evaluate the performance of a search in the context of 'extreme' examples. Although very small with respect to lines of code, the input domain for the synthetic programs ranges up to 10^{10}. The degree of difficulty for search–based testing is determined by the size of the search–space as well as the shape of the fitness landscape, ensuring the synthetic examples are not trivial. In addition, the dynamic memory allocation was designed to add further complexity to the problem.

A search was terminated if either:

1. an ideal solution was found, or
2. 100,000 fitness evaluations had been performed, or
3. no progress, with respect to the current best solution, had been made over 25 generations.

For the Pareto GA, an ideal solution was considered to be the best solution with respect to memory allocation, which also achieved the branch target.

6.5 Case Study Results

Each case study consists of the three algorithms run 10 times. These results are presented in Figure 6.1.

Case Study 1 is the `addscan` function from the `space` program, used by the European Space Agency for scanning star field patterns. Memory is allocated without releasing it, thus to avoid memory leaks, the function was modified for test purposes to free the allocated memory.

Summary: Overall the Pareto GA allocates 66% more bytes than the weighted GA at the expense of branch coverage. The function contains 32 branches and its

domain size is approximately 10^{539}. The weighted GA is deliberately directed towards covering branches, making it more likely to succeed within the limits set by the stopping conditions described above. The 3 branches left uncovered were either infeasible or the search simply failed.

In order for the Pareto GA to cover a branch it needs to find an 'extreme' point on the Pareto front. These points will only be 'discovered' quite late in the search, because a Pareto GA always tries to find a good spread of solutions across the front, evolving from a central region towards the 'end points' of a frontline.

Case Study 2 is a function taken from the `cgi-util.c` source, which is based on `post-query.c` and `query.c` by NCSA. It takes a string and a 'stop' character as input, and parses the string until either a terminating null or a stop character is found. The new substring is removed from the input string and returned by the function. Only the length of the input string affects the amount of memory being allocated. However, this amount is constant for all branches because the memory allocation occurs at the start of the function. The length of the input string was restricted to 10^4 characters for practical reasons. This function also had to be modified to release the memory allocated after each test run to prevent memory leaks.

Summary: The results from Figure 6.1 confirm that a random search is good at achieving high branch coverage for 'easy to cover' branches. However, over 10 runs the search only manages to allocate 13.62% of the optimum for memory allocation. The weighted GA used a weight distribution of 3 : 2 in favour of branch coverage for this case study. It manages to comprehensively beat the Pareto GA in terms of branch coverage. However, over 10 runs the weighted GA allocates fewer bytes (6258.7 on average) than the Pareto GA. While the Pareto GA manages to allocate the maximum amount of memory possible in 1 run, its average branch coverage is only 95%. Thus, the weighted GA can be considered better suited for this case study because achieving full branch coverage is required.

Case Study 3 is a function containing four predicates. The first three follow an `if - then - else - if` structure and allocate a constant amount of memory. The true branch of the first predicate allocates 20 bytes, the true branch of the second predicate 10 bytes and the third predicate, 5 bytes. The last predicate does not allocate any memory. This example was chosen to investigate an inverse relationship between

'approach level' for branch coverage and memory allocation and the effects on finding a Pareto optimal set.

Summary: The random search is uninteresting as it neither achieves 100% coverage nor allocates any memory at all in 9 of the 10 runs. The Pareto GA finds the maximum amount of bytes that can be allocated in 50% of the runs. However, it also leaves at best 3 branches uncovered. By achieving 100% branch coverage, the weighted GA manages to optimize the input vector to allocate the maximum amount of memory possible in all runs, clearly outperforming the Pareto GA at less computational cost.

Case Study 4 is a program that generates a random sequence of characters from the alphabet and stores them in a string. The length of the string generated depends on the input parameters. The first parameter affects the path to be taken through the function; the second parameter specifies the length of the string. The branching nodes ensure that the second parameter only influences the memory allocation if the test case traverses the true branch of the first predicate. If a test case fails to cover this branch, the memory allocation for the rest of the function will either be constant, or, in one case, the function will exit prematurely and no memory will be allocated.

Summary: The input domain for Case Study 4 is 10^5 and the maximum number of bytes that can be allocated is restricted to 256. Even though the input domain is quite small, all methods fail to cover 100% of the branches. One of the uncovered branches is controlled by a predicate checking if the memory allocation was successful. The true branch of this predicate can be considered infeasible because of the restrictions imposed on the memory allocation. All other branches are covered by the weighted GA 40% of the time. Both the Pareto GA and random search fail to cover at least one more branch than the weighted GA. One of these uncovered branches is controlled by a flag–containing predicate. The success of the weighted GA in covering this branch can be explained by the distribution of weights. These ensure more resource is spent on the branch coverage objective compared to the Pareto GA, which shares its resources amongst the objectives.

Case Study 5 is a function constructed to produce a 'difficult to search' fitness landscape for the GAs (MBHar). To further add complexity, the memory allocation is constant for all but one branch. The false branch of a 'hard to cover' predicate contains an additional reallocation of memory, which in effect 'rewards' a search for missing the target, creating a deliberate conflict between the two objectives.

Summary: The random search fails to allocate more than 33% of the total possible memory and only covers 17% of all branches. Perhaps surprisingly the Pareto GA outperforms the weighted GA by covering at least 1 more branch than the weighted GA in all but 1 run. It also consistently allocates the maximum number of bytes possible over the 10 runs, whereas the weighted GA only manages to do so in 50% of the runs.

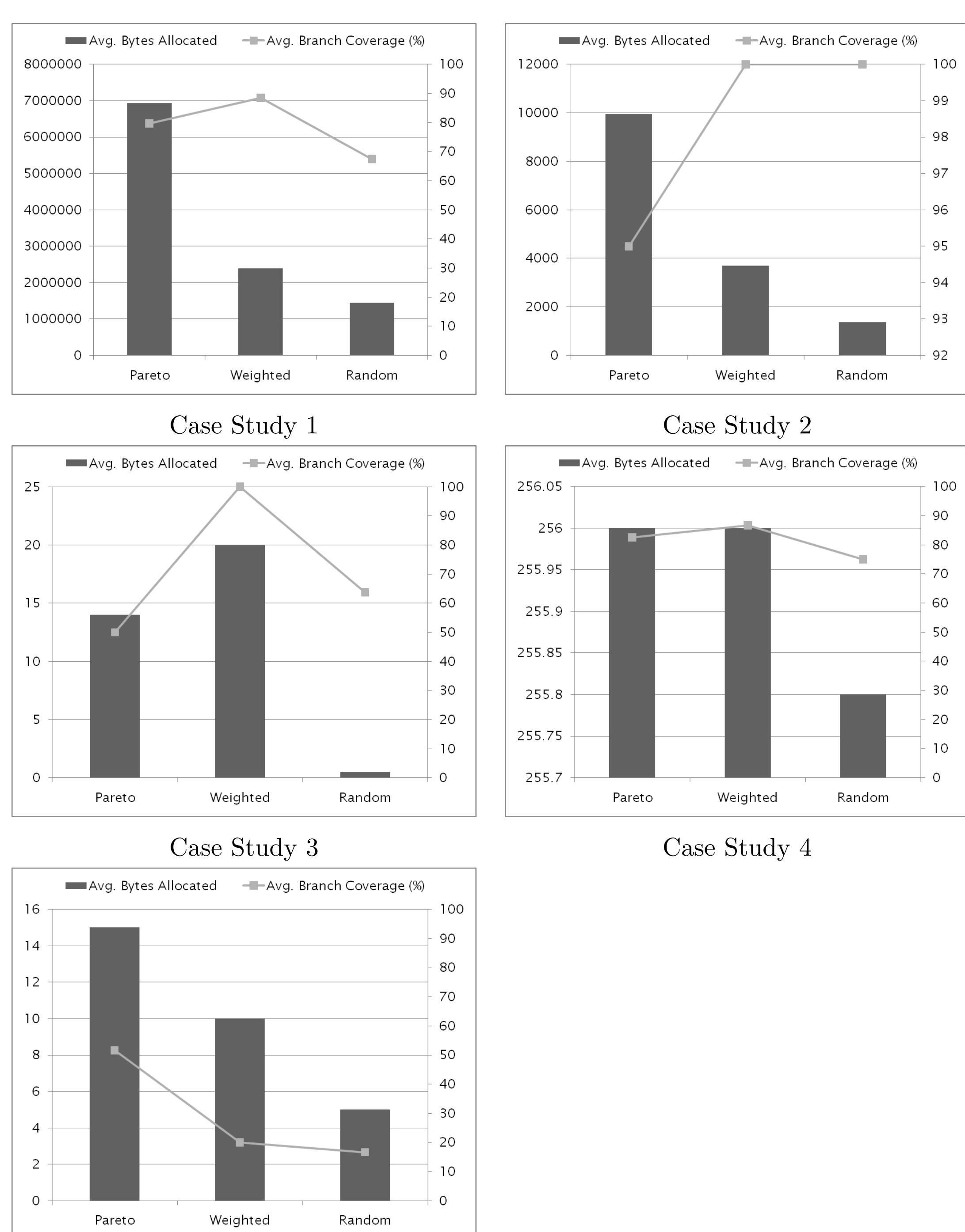
6.6 Discussion

This section discusses the findings of the case studies from Section 6.5 and the results presented in Figure 6.1. It also includes some insights gained during the case studies.

For Case Studies 3 and 4 the weighted GA clearly outperforms the Pareto GA in all objectives. Case Study 1 presents a trade–off between the two GAs. The weighted GA covers an average 88% of the branches, compared to just under 80% covered by the Pareto GA. However, the weighted GA only manages to allocate about 34% of the number of bytes the Pareto GA allocates over 10 runs. In Case Study 2 the weighted GA achieves an overall higher branch coverage than the Pareto GA, while allocating fewer bytes. Finally, in Case Study 5 the Pareto GA beats the weighted GA over a total of 10 runs.

The maximum number of bytes recorded during the case studies refer to the highest values found by a test case which covered a particular target. Any 'better' value found for this objective whilst attempting the target was not recorded if the test case missed the target and covered another branch, *e.g.* branch b instead. However, Figure 6.2 shows that the ideal solution for branch b will allocate at least the same amount of memory.

Overall, the findings suggest that it is not possible to pick one search method over the other, as each performs better in some cases. For example, Table 6.1 illustrates that both the weighted and Pareto GA, cover branches missed by either Pareto or weighted GA respectively. Equally, both approaches have a number of disadvantages: the high computational cost associated with the Pareto GA and the difficulty of finding the most efficient set of weights for the weighted GA. For some example functions even a slight increase in the weight for the memory objective resulted in a significant drop in branch coverage. For others, adjusting the weights did not seem to affect the branch



Case Study 5

Figure 6.1: Results of the branch coverage and memory allocation achieved by three different algorithms: a random search, a Pareto GA and, weighted GA.

coverage and only marginally improved the memory allocation, even with a 'drastic' redistribution of weights. A set of experiments were carried out to investigate the impact different weights have on the behaviour of the weighted GA before deciding on the final weights used in the case studies.

Given the results presented in Table 6.1 and Figure 6.1, a hybrid approach may be advisable. For example, a weighted GA could be used to cover branches for which the Pareto GA failed to find test cases. This would also solve the issue of not being able to evaluate the performance of a weighted GA for an objective with an undefined optimum. Case Study 1 is an example where the maximum number of bytes allocated by the weighted GA is meaningless without a point of reference. The Pareto GA is more likely to find a good approximation to the 'real' optimum because it has less room for error, *e.g.* by not having an ideal distribution of weights.

The case studies also revealed that, in most cases, the Pareto GA does not produce a frontline. It converges at a single solution instead (see Figure 6.3). Where a frontline exists, it often lacks diversity. While this is not entirely due to the sub–goal approach, it is emphasized by it.

For a frontline to contain many points, the target branch needs to either allocate varying amounts of memory, or an inverse relationship between the distance of a test case from the target and the memory allocated by it, must exist (see Figure 6.2).

In any other scenario the frontline contains at most two points: one point representing a case that reached the target and the other, any test case that happens to allocate more memory than the first but misses the target.

Another issue revealed is that 100% branch coverage is very hard to achieve for programs containing `malloc`, `calloc` and `realloc` statements. After allocating memory, a well written program should check whether the allocation has been successful. It is these cases that are of interest, and which partly motivated the exploration of applying a Pareto GA to the branch coverage problem. However, to exhaust a program's heap space can be very challenging, possibly requiring a large number of loop–iterations or the presence of a memory leak to name but a few scenarios. Once the heap space is exhausted, the C program may crash, especially if it is not well written. As a result, the test environment will also terminate, thus being unable to log the test case that caused the crash.

```
int p*;
if( a == 0)
{
    /*target 1T*/
}
else
{
   /*target 1F*/
   p = (int*)malloc(a*sizeof(int));
}
```

Code snippet

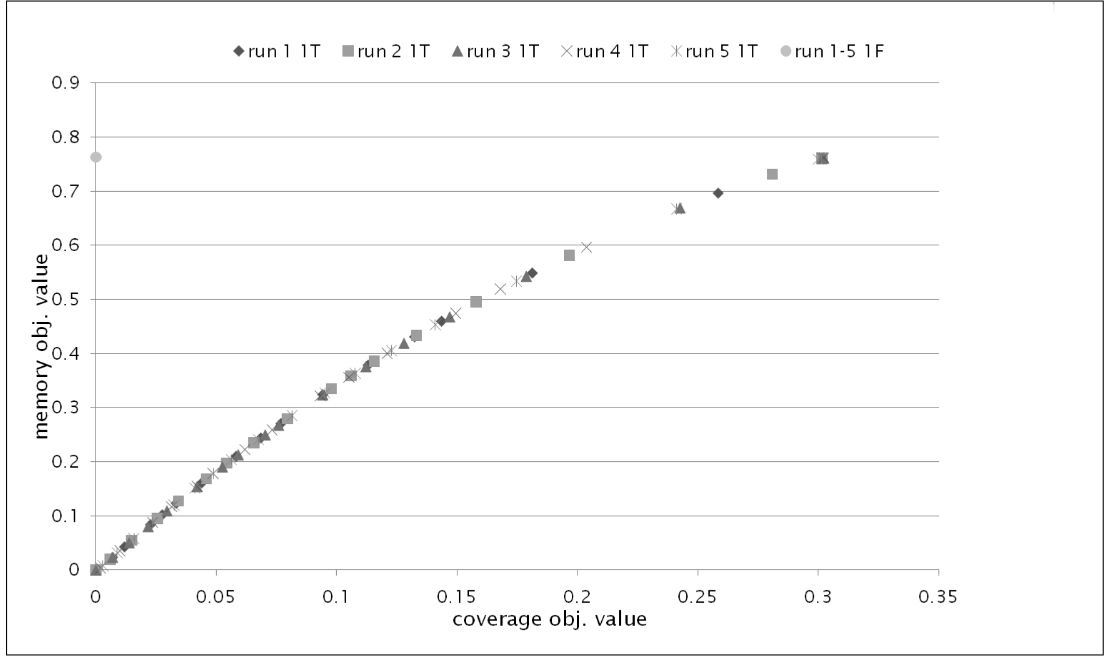


Figure 6.2: Final Pareto fronts produced for targets 1T and 1F. The upper point on the y–axis represents the 'ideal' solution for target 1F. As can be seen, once the branch has been reached, a single solution will dominate all others because it is the only branch allocating memory. When attempting to cover target 1T on the other hand, the Pareto optimal set potentially consists of an infinite number of solutions. The graph combines five runs which reveal little variance between the frontlines produced. Interestingly, the 'ideal' point for target 1F corresponds to the maximum value contained within the Pareto optimal set for target 1T with respect to memory allocation.

Branch ID	Example Function
	`char *makeword(char *line, char stop) {`
	`  int x = 0,y;`
	`  char *word;`
	`  word=(char *)malloc(sizeof(char)*(strlen(line)+ 1));`
1T/1F	`  for(x=0;((line[x]) && (line[x] != stop));x++)`
	`    word[x] = line[x];`
	`  word[x] = '\0';`
2T/2F	`  if(line[x]) ++x;`
	`    y=0;`
3T/3F	`  while(line[y++] = line[x++]);`
	`  return word;`
	`}`

	Bytes allocated
1T	{9634}
1F	{9856}
2T	{9692}
2F	{1194}
3T	{9553}
3F	{9649}

Figure 6.3: The table at the bottom presents the Pareto optimal sets for each 'sub–goal' of the example function above used in Case Study 2. It combines the results collected over five runs and illustrates that it is often not possible to generate a Pareto frontline when considering branch coverage and memory allocation as a MOP. Although the amount of dynamic memory allocated depends on the input parameters, it is constant for all branches. As a result one solution will dominate all others with respect to a particular target.

Table 6.1: The table shows the branches covered by the weighted GA and not the Pareto GA, or vice versa. The 'distance' measure illustrates how close the best solution came to traversing the target branch. It combines the normalized branch distance and the approach level. A distance of 0 indicates a branch has been covered. These results were obtained during Case Study 1.

Branch ID	Bytes allocated		Distance	
	Pareto	weighted	Pareto	weighted
7	2826560	690360	0.001997004	0
9	2494888	534160	0.005979056	0
10	5622848	1529968	1.001997004	0
11	5978016	1309352	2.001997004	0
12	6372608	586784	2.001997004	0
15	6969776	518320	0.000699161	0
16	1374560	1304160	0.5	0
17	2644048	758032	1.000999001	0
21	2455024	195888	3.000999001	0
28	6728128	291368	0	6.2

6.7 Conclusion

This chapter has presented a first multi–objective approach to branch coverage. Traditionally, the aim of branch coverage has solely been to find test cases which traverse a specific branch. The chapter supplements this goal with the additional objective of consuming as much dynamic memory as possible at the same time.

Five case studies, two based on real–world C code and three created to push the techniques to the extremes, compared the performance of three search methods: a random search, Pareto GA and weighted GA. The results show that a weighted GA is best suited in most cases, achieving the same results as a Pareto GA more efficiently. However, the studies also reveal that a hybrid approach between the two algorithms may offer the best overall results.

Chapter 7

Conclusions and Future Work

7.1 Summary of Achievements

This thesis has proposed two solutions to extend and improve search–based testing. In addition it has provided a first formulation of a multi–objective branch coverage test adequacy criterion. The original aims and objectives of the thesis were as follows:

1. To advance the capabilities of the current state–of–the–art search–based testing techniques, extending them so they can handle pointers and dynamic data structures.
2. To perform a thorough empirical investigation, evaluating the extended search–based strategy against a concolic testing approach. The study aimed to provide a concrete domain of programs for which the approaches will either be adequate or inadequate.
3. To empirically investigate the use of a testability transformation for the loop–assigned flag problem in search–based–testing.
4. To investigate the use of search–based testing in multi–objective test data generation problems.

The first of the objectives was addressed by combining a constraint solving method based on symbolic execution in a novel way with a hill climber in a tool called AUSTIN. The tool was then evaluated in an empirical study in which its effectiveness and efficiency in generating test data were compared with that of the state–of–the–art ETF

structural test component, developed within the scope of the EU–funded EvoTest project. A comparison was also made with the ETF configured to perform a random search. The test objects consisted of eight non–trivial C functions drawn from three real–world embedded software modules from the automotive sector and implemented using two popular code–generation tools. For the majority of the functions, AUSTIN is at least as effective (in terms of achieved branch coverage) as the ETF and is considerably more efficient. These findings agree with previous research comparing the performance of a hill climbing algorithm with a genetic algorithm for structural test data generation on a smaller scale. The empirical study presented in Chapter 3 also goes some way towards addressing the lack of 'real–world' evaluations in search–based testing.

After successfully applying AUSTIN to machine generated code and achieving relatively high levels of coverage, the second objective was achieved by comparing AUSTIN with CUTE, a concolic testing tool, on five large open source applications. The study is the first of its kind. Each tool was applied 'out of the box'; that is, without writing additional code for special handling of any of the individual subjects, or by tuning the tools' parameters. Perhaps surprisingly, the results show that both tools can only obtain, at best, a modest level of code coverage. For AUSTIN the two major contributors to its poor performance were segmentation faults in the function under test and spikes and plateaux in the fitness landscape. Segmentation faults were caused by the absence of preconditions for pointer inputs. AUSTIN currently cannot infer such preconditions, *i.e.* it either requires a tester to specify preconditions in the test driver, or, requires a program to be coded 'defensively'. A defensive programming style includes checks on pointer variables to ensure they are initialized before dereferencing their value. The second contributor was tackled in Chapter 5 by proposing a testability transformation for loop–assigned flags.

Most search–based testing approaches degenerate to random testing in the presence of flag variables, because flags create spikes and plateaux in the fitness landscape. Both these features are known to denote difficult optimization problems for search–based optimization techniques. Several authors have studied flag removal transformations and fitness function refinements to address the issue of flags, but the problem of loop–assigned flags remained unsolved. This thesis introduced a testability transformation

along with a tool that transforms programs with loop–assigned flags into flag–free equivalents, so that existing search–based test data generation approaches can successfully be applied.

The behaviour of the algorithm was evaluated with two empirical studies that involved a synthetic program and functions taken from open source programs. The synthetic examples were used to illustrate how two variations of the algorithm perform for different levels of difficulty. The results show that the approach scales well to very difficult search landscapes, for which test data is notoriously hard to find. The worst–case considered involves finding a single adequate test input from a search–space of size 2^{320}. Despite the size and difficulty of this search problem, the search–based testing approach, augmented with the transformation algorithm introduced in this thesis, consistently finds this value.

The final objective of the thesis was achieved by investigating a multi–objective branch coverage problem. Traditionally, the aim of branch coverage has solely been to find test cases which traverse a specific branch. However, in many scenarios a single–objective formulation is unrealistic; testers will want to find test sets that meet several objectives simultaneously in order to maximize the value obtained from the inherently expensive process of running the test cases and examining the output they produce. In Chapter 6 the thesis supplemented the goal of generating branch adequate test data with the additional objective of consuming as much dynamic memory as possible at the same time. This new multi–objective problem was investigated with the help of five case studies, two based on real–world C code and three created to push the techniques to the extremes. Three search methods were used during the evaluation: a random search, Pareto GA and weighted GA. The results show that a weighted GA is best suited in most cases, achieving the same results as a Pareto GA more efficiently.

In summary, Chapters 3, 4 and 6 are novel to this thesis, while the work in Chapter 5 is an extension of previous work by Baresel *et al.* (BBHK04). The work in this thesis differs from the work of Baresel *et al.* in the following primary ways:

1. modified flag replacement algorithm;
2. new implementation section;
3. empirical evaluation of the flag replacement algorithm.

The work in Chapter 5 also forms part of a recently published article in Transactions on Software Engineering and Methodology (BHLar). The published article also contains an empirical study to show that the flag problem considered is prevalent among those uses of flags found in a suite of real–world programs. This empirical study does not feature in Chapter 5.

7.2 Summary of Future Work

Despite the promising results obtained by AUSTIN in Chapter 3, bridging the gap between research and industrial applicability remains an open problem for search–based testing tools. For example, search–based testing tools targeting C code need to support a wider range of input types, such as union constructs, void pointers, function pointers and strings. While work has been done to support string inputs in testing, mostly outside the field of search–based testing, no previous work in the field of search–based testing addresses the problem of void and function pointers in C code, even though they are commonly used programming paradigms in real–world code. Also, on a practical level, tools need to be able to prevent or recover from segmentation faults, so that they may continue the test data generation process after the system under test raises such exceptions.

Any future work will also have to consider a large body of real–world code. In order to extend the scalability of AUSTIN further, the path condition collected during the symbolic execution (described in Chapter 3) will be used to perform an input domain reduction. This will be done in two stages. First, a constraint solver will be used for any linear constraints in order to improve the efficiency of the search by saving hill climb moves. The intention is to use a hill climb algorithm only in the presence of floating point calculations, non–linear expression, or generally for branch predicates which are too complex to be handled by a lightweight constraint solver. Suppose execution of the path which executes the true branch of the program of Figure 7.1(a) is required. AUSTIN will start by executing the function with the values `0` and `0` for `x` and `y` respectively. It will then take AUSTIN a total of 5 fitness evaluations to reach the true branch depicted in Figure 7.1(a). By contrast, a dynamic symbolic execution based tool would be able to do so in a maximum of 2 iterations (*i.e.* executions of the function). AUSTIN already forms a path condition for all symbolic variables,

but currently discards constraints which are not over pointer inputs. It would not take much to filter out constraints over primitive type variables and pass those to a constraint solver. Further, it would enable AUSTIN to benefit from another strength of symbolic execution, namely being able to detect infeasible paths, and thus save wasteful fitness evaluations.

The second stage is to perform a dynamic variable dependence analysis in order to reduce the size of the search–space. Suppose execution of the path which executes the true branch of the program of Figure 7.1(b) is required again. AUSTIN will start by executing the function with all variables of primitive type set to zero. When starting its exploratory moves, it will waste 4 moves by exploring neighbouring values for the inputs `a` and `b` respectively. These wasteful moves will be repeated every time a pattern move finishes, or a random restart is performed. The *un–trimmed* path condition AUSTIN constructs (see Section 3.2.5) for the path executed by input variables set to zero is $\langle x * y \geq 100 \rangle$. The proposed idea is to only include those variables in the AVM which appear in constraints in the path condition, *i.e.* `x` and `y`, thereby saving potentially wasteful moves. The dynamic input domain reduction would be more light-weight than a more traditional static analysis such as the one used by Harman *et al.* (HHL$^+$07). Further, it is currently not straightforward to include the results of the static analysis in the test data generation process, requiring the assistance of a human. The idea presented in this thesis is fully automated.

Once the (predominantly) implementation related problems have been solved, and AUSTIN has been made more efficient, the next challenge will be to improve the quality of test data produced by an automated technique. To date, the quality of a technique for generating test data is still overwhelmingly based on the number of syntactic features covered, *e.g.* branches in a function. Yet, there has been much debate about whether or not these measures correlate to fault revelation (MRBW95; Bei90). Furthermore, the current techniques do not consider the *oracle cost*; the time and effort it takes (for a human) to check that the input values generated by the automated technique

1. obey any implied precondition to the function
2. did not produce an unexpected behaviour of the function under test
3. resulted in the expected output values given the input values

```
void testme1(int a, int b)
{
    a += 5; b -= 10;
    if (a == b)
        // ...
}
```

(a) Example for demonstrating how AUSTIN 'wastes' fitness evaluations. The branch predicate is linear.

```
void testme2(int a, int b, int x, int y)
{
    if (x * y < 100)
        // ...
}
```

(b) Example for demonstrating how AUSTIN could use a light weight dynamic input domain reduction. The branch predicate is non–linear.

Figure 7.1: Code example to illustrate ideas for future work.

The above points are inherently hard to automate, especially for programming languages like C, which do not natively support features such as explicit pre and post conditions of a function. One possible way to strengthen the test data produced by an automated technique could be to link the test data generation process with an automated invariant inference technique.

The most popular invariant detector is the Daikon tool (EPG$^+$07). Daikon is able to generate program invariants based on program trace files. These files are produced by executing a set of test cases (inputs to a program) and recording their effect. The 'quality' of the inferred invariants are thus largely dependent on the quality of the original test set which was used to obtain the invariants.

One way to improve both the quality of the invariants as well as the test data would be to use the test data generation as a means to generate counter examples to drive the development of new pre– and post–conditions. In this way both the specifications and the test cases would form a kind of co–evolution. Initially, a test set would consist of branch adequate test data, *i.e.* test data search–based techniques such as those implemented in AUSTIN are currently able to generate. Suppose that the test set has

a 'poor' quality according to a fault revelation metric. Any inferred specifications from such a test set will consequently also be of poor quality, because they are likely to be inaccurate. However, as the test cases 'evolve' and reveal counter examples, the specification inference will become ever more precise, as the pool of data from which observations are made increases.

The end result is a set of test data and a specification for the function under test. Even if the test cases obtained through this co–evolution do not turn out to be more fault revealing than test data produced by existing techniques, they are still likely to yield deep insights into the relationship between test cases and specifications. Further, they should reduce the burden on testers by presenting a specification which is close to the true specification of the function, and a set of test cases which should, in theory, be easier to validate because they are coupled with a specification.

Appendix A

Abbreviations and Acronyms

API – Application Programming Interface

AUSTIN – AUgmented Search–based TestINg

AVM – Alternating Variable Method

CFG – Control Flow Graph

CIL – C Intermediate Language

ETF – Evolutionary Testing Framework

EU – European Union

FUT – Function Under Test

GA – Genetic Algorithm

IGUANA – Input Generation Using Automated Novel Algorithms

LOC – Lines Of Code

MC/DC – Modified Condition / Decision Coverage

MOEAs – Multi–Objective Evolutionary Algorithms

MOP – Multi–Objective Problem

SBST – Search–Based Software Testing

References

[AB06] Mohammad Alshraideh and Leonardo Bottaci. Using Program Data-State Diversity in Test Data Search. In *Proceedings of the 1st Testing: Academic & Industrial Conference - Practice and Research Techniques (TAICPART '06)*, pages 107–114, Cumberland Lodge, Windsor, UK, 29-31 August 2006. IEEE. (Cited on pages 122 and 138.)

[AC05] Enrique Alba and Francisco Chicano. Software Testing with Evolutionary Strategies. In *Proceedings of the 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE '05)*, volume 3943 of *Lecture Notes in Computer Science*, pages 50–65, Heraklion, Crete, Greece, 8-9 September 2005. Springer. (Cited on page 17.)

[AC08] Enrique Alba and Francisco Chicano. Observations in using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing. *Computers & Operations Research*, 35(10):3161–3183, October 2008. (Cited on page 17.)

[ASC] ETAS ASCET. `http://www.etas.com/en/products/ascet_software_products.php`. (Cited on page 72.)

[AY08] Andrea Arcuri and Xin Yao. Search Based Software Testing of Object-Oriented Containers. *Information Sciences*, 178(15):3075–3095, August 2008. (Cited on page 17.)

[Bak87] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Genetic Algorithms and their Applications (ICGA'87)*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates. (Cited on page 145.)

[BBHK04] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary Testing in the Presence of Loop-Assigned Flags: A Testability Transformation Approach. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 108–118, Boston, Massachusetts, USA, 11-14 July 2004. ACM. (Cited on pages xiv, 123, 124, 125, 126 and 159.)

[Bei90] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990. (Cited on page 161.)

[BFJT05] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test, Verif. Reliab*, 15(2):73–96, 2005. (Cited on page 2.)

[BGM06] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test, Verif. Reliab*, 16(2):97–121, 2006. (Cited on pages 26 and 27.)

[BHLar] Dave Binkley, Mark Harman, and Kiran Lakhotia. FlagRemover: A Testability Transformation For Loop Assigned Flags. *Transactions on Software Engineering and Methodology*, To appear. (Cited on pages xiv, 110, 112 and 160.)

[BJ01] Paulo Marcos Siqueira Bueno and Mario Jino. Automatic Test Data Generation for Program Paths Using Genetic Algorithms. In *Proceedings of the 13th International Conference on Software Engineering & Knowledge Engineering (SEKE '01)*, pages 2–9, Buenos Aires, Argentina, 13-15 June 2001. (Cited on page 16.)

[Bot02] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, New York, 9-13 July 2002. Morgan Kaufmann Publishers. (Cited on pages 110, 112, 113, 115, 122 and 137.)

[Bri98] British Standards Institute. BS 7925-2 software component testing, 1998. (Cited on pages 2, 16 and 109.)

[BS03] André Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation (GECCO-2003)*, volume 2724 of *LNCS*, pages 2442–2454, Chicago, 12-16 July 2003. Springer-Verlag. (Cited on pages 110, 112, 113, 122, 137 and 138.)

[BS08] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446. IEEE, 2008. (Cited on pages 30, 31, 73, 90 and 107.)

[BSS02] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336, San Francisco, CA 94104, USA, 9-13 July 2002. Morgan Kaufmann Publishers. (Cited on page 123.)

[BTDD07] Raquel Blanco, Javier Tuya, Eugenia Daz, and B. Adenso Daz. A Scatter Search Approach for Automated Branch Coverage in Software Testing. *International Journal of Engineering Intelligent Systems (EIS)*, 15(3):135–142, September 2007. (Cited on page 17.)

[Cat05] Nestor Cataño. Combining symbolic execution and model checking to reduce dynamic program analysis overhead. *Revista Comlombiana de Computación*, 6(1), 2005. (Cited on page 25.)

[CDH+03] John Clark, José Javier Dolado, Mark Harman, Robert Mark Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, and Martin Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003. (Cited on page 2.)

[CE05] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23. Springer, 2005. (Cited on pages 2, 3, 27, 29, 31, 35, 87 and 90.)

[CG06] Arindam Chakrabarti and Patrice Godefroid. Software partitioning for effective automated unit testing. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT 2006, October 22-25, 2006, Seoul, Korea*, pages 262–271. ACM, 2006. (Cited on pages 31 and 108.)

[CGMC03] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 38–48, Piscataway, NJ, May 3–10 2003. IEEE Computer Society. (Cited on page 2.)

[CK06] Yoonsik Cheon and Myoung Kim. A Specification-based Fitness Function for Evolutionary Testing of Object-oriented Programs. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1953–1954, Seattle, Washington, USA, 8-12 July 2006. ACM. (Cited on page 17.)

[CPDGP01] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying safety-critical systems. In *European Software Engineering Conference (ESEC) / ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 142–151, 2001. (Cited on page 25.)

[CS89] Rich Caruana and J. David Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the 5th International Conference on Machine Learning*, Los Altos, CA, June 1989. Morgan Kaufmann Publishers. (Cited on page 18.)

[CS09] Luís Da Costa and Marc Schoenauer. Bringing evolutionary computation to industrial applications with guide. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 1467–1474. ACM, 2009. (Cited on page 70.)

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 281–290. ACM, 2008. (Cited on page 90.)

[DAPM00] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Parallel Problem Solving from Nature – PPSN VI*, pages 849–858, Berlin, 2000. Springer. (Cited on pages 143, 146 and 147.)

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification, 18th International Conference, CAV 2006,*

Seattle, WA, USA, August 17-20, 2006, Proceedings, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006. (Cited on page 27.)

[DM77] Nachum Dershowitz and Zohar Manna. The evolution of programs: A system for automatic program modification. In *Conference Record of the Fourth Annual Symposium on Principles of Programming Languages*, pages 144–154. ACM SIGACT and SIGPLAN, ACM Press, 1977. (Cited on page 139.)

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, 2008. Springer-Verlag. (Cited on page 27.)

[dT] dSpace TargetLink. `http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm`. (Cited on page 72.)

[DTBD08] Eugenia Díaz, Javier Tuya, Raquel Blanco, and José Javier Dolado. A Tabu Search Algorithm for Structural Software Testing. *Computers & Operations Research*, 35(10):3052–3072, October 2008. (Cited on page 17.)

[EPG+07] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007. (Cited on page 162.)

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982. (Cited on page 139.)

[FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996. (Cited on pages 15, 106, 110 and 137.)

[FV95] Mark A. Friedman and Jeffrey M. Voas. *Software Assessment: Reliability, Safety, Testability.* John Wiley & Sons, 1995. (Cited on page 112.)

[GdHN+08] Patrice Godefroid, Jonathan de Halleux, Aditya V. Nori, Sriram K. Rajamani, Wolfram Schulte, Nikolai Tillmann, and Michael Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008. (Cited on page 26.)

[GK04] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):117–127, 2004. (Cited on page 139.)

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *ACM SIGPLAN Notices*, 40(6):213–223, June 2005. (Cited on pages 2, 3, 27, 28, 31, 35, 37 and 87.)

[GKWV09] Hamilton Gross, Peter M. Kruse, Joachim Wegener, and Tanja Vos. Evolutionary white-box software test with the evotest framework: A progress report. In *ICSTW '09: Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on pages 36, 70, 71 and 75.)

[God07a] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54. ACM, 2007. (Cited on pages 90 and 108.)

[God07b] Patrice Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *RT '07: Proceedings of the 2nd international workshop on Random Testing*, pages 1–1, New York, NY, USA, 2007. ACM. (Cited on pages 90 and 108.)

[gre] GNU grep. `http://www.gnu.org/software/grep/`. (Cited on page 107.)

[Har07] Mark Harman. The current state and future of search based software engineering. In *IEEE Computer Society Press*, Future of Software Engineering 2007, Los Alamitos, California, USA, 2007, 2007. (Cited on page 2.)

[HBC+04] Arturo Hernández Aguirre, Salvador Botello Rionda, Carlos A. Coello Coello, Giovanni Lizárraga Lizárraga, and Efrén Mezura Montes. Handling Constraints using Multiobjective Optimization Concepts. *International Journal for Numerical Methods in Engineering*, 59(15):1989–2017, April 2004. (Cited on page 143.)

[HC04] Mark Harman and John Clark. Metrics are fitness functions too. In 10th *International Software Metrics Symposium (METRICS 2004)*, pages 58–69, Los Alamitos, California, USA, September 2004. IEEE Computer Society Press. (Cited on page 16.)

[HHH+04] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, January 2004. (Cited on pages 103, 110, 112, 113, 122, 137 and 138.)

[HHL+07] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 155–164. ACM, 2007. (Cited on pages 105 and 161.)

[HM09] Mark Harman and Phil McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 2009. to appear. (Cited on pages 2, 17, 37, 74, 75, 78, 86, 87, 90 and 124.)

[HR] Mary Jean Harrold and Gregg Rothermel. `http://pleuma.cc.gatech.edu/aristotle/Tools/subjects/Tools/subjects`. (Cited on page 107.)

[IX07] Kobi Inkumsah and Tao Xie. Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 425–428, November 2007. (Cited on pages ix, 32 and 33.)

[JM81] Neil D. Jones and Steven S. Muchnick. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981. (Cited on page 25.)

[JSE96] Bryan F. Jones, Harmen H. Sthamer, and David E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996. (Cited on page 22.)

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. (Cited on pages 2, 3, 25 and 26.)

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990. (Cited on pages xi, xvii, 2, 3, 13, 14, 17, 22, 35, 59 and 90.)

[Kor92] Bogdan Korel. Dynamic method of software test data generation. *Softw. Test, Verif. Reliab*, 2(4):203–213, 1992. (Cited on pages ix, 15 and 16.)

[LBW04] Frank Lammermann, André Baresel, and Joachim Wegener. Evaluating Evolutionary Testability with Software-Measurements. In *Proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO '04)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1350–1362, Seattle, Washington, USA, 26-30 June 2004. Springer Berlin / Heidelberg. (Cited on page 16.)

[LHM08] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling Dynamic Data Structures in Search Based Testing. In Maarten Keijzer, editor, *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08)*, pages 1759–1766, Atlanta, GA, USA, 12-16 July 2008. ACM. (Cited on page 16.)

[LI08] Raluca Lefticaru and Florentin Ipate. Functional Search-based Testing from State Machines. In *Proceedings of the First International Conference on Software Testing, Verfication and Validation (ICST 2008)*, pages 525–528, Lillehammer, Norway, 9-11 April 2008. IEEE Computer Society. (Cited on page 17.)

[Lit01] Tim Littlefair. *An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment.* PhD thesis, Faculty of computing, health and science, Edith Cowan University, Australia, 2001. (Cited on pages xvii, 72 and 74.)

[LLLW05] Xiyang Liu, Ning Lei, Hehui Liu, and Bin Wang. Evolutionary testing of unstructured programs in the presence of flag problems. In *APSEC*, pages 525–533. IEEE Computer Society, 2005. (Cited on pages 122, 137 and 138.)

[LLW+05] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *20th IEEE/ACM*

International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA, pages 337–341. ACM, 2005. (Cited on page 138.)

[LMH09] Kiran Lakhotia, Phil McMinn, and Mark Harman. Automated Test Data Generation for Coverage: Haven't We Solved This Problem Yet? In *4th Testing Academia and Industry Conference - Practice and Research Techniques TAIC PART 09*, pages 95–104, 2009. (Cited on pages x, 37 and 40.)

[MBHar] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Transactions on Software Engineering Methodology*, To appear. (Cited on page 150.)

[McM04] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004. (Cited on pages 2 and 16.)

[McM05] Phil McMinn. *Evolutionary Search For Test Data In The Presence Of State Behaviour*. PhD thesis, University of Sheffield, 2005. (Cited on page 24.)

[McM07] Phil McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical Report CS-07-14, Department of Computer Science, University of Sheffield, 2007. (Cited on page 145.)

[MH06] Phil McMinn and Mike Holcombe. Evolutionary testing using an extended chaining approach. *Evolutionary Computation*, 14(1):41–64, 2006. (Cited on pages x, 16, 40, 106 and 137.)

[MHH01] Rob Hierons Mark Harman and Lin Hu. Testability enhancement transformation. Technical report, DAIMLER-CHRYSLER, 2001. (Cited on page 112.)

[MMS01] Christoph C. Michael, Gary McGraw, and Michael Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng*, 27(12):1085–1110, 2001. (Cited on pages 2, 16 and 108.)

[MRBW95] James Miller, Marc Roper, Andrew Brooks, and Murray Wood. Towards a benchmark for the evaluation of software testing techniques. *Information and Software Technology*, 37(1):5–13, 1995. (Cited on page 161.)

[MRZ06] James Miller, Marek Reformat, and Howard Zhang. Automatic Test Data Generation using Genetic Algorithm and Program Dependence Graphs. *Information and Software Technology*, 48(7):586–605, July 2006. (Cited on page 16.)

[MS76] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, September 1976. (Cited on pages 1, 2 and 13.)

[MS07] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, pages 416–426. IEEE Computer Society, 2007. (Cited on pages 90 and 107.)

[MSV93] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm: I. continuous parameter optimization. *Evolutionary Computation*, 1(1):25–49, 1993. (Cited on page 145.)

[MT81] Silvano Martello and Paolo Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27(2):93–112, 1981. (Cited on page 143.)

[NMRW02] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002. (Cited on pages 41 and 118.)

[OJP99] A. Jefferson Offutt, Z. Jin, and Jie Pan. The dynamic domain reduction approach to test data generation. *Software Practice and Experience*, 29(2):167–193, January 1999. (Cited on page 137.)

[PHP99] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Softw. Test, Verif. Reliab*, 9(4):263–282, 1999. (Cited on pages 2, 16 and 22.)

[PW08] Maria Prutkina and Andreas Windisch. Evolutionary Structural Testing of Software with Pointers. In *Proceedings of 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008*, pages 231–231, Lillehammer, Norway, 9-11 April 2008. IEEE. (Cited on pages 16 and 70.)

[Rad92] Radio Technical Commission for Aeronautics. RTCA DO178-B Software considerations in airborne systems and equipment certification, 1992. (Cited on pages 2, 16 and 109.)

[RMB+95] Marc Roper, Iain Maclean, Andrew Brooks, James Miller, and Murray Wood. Genetic algorithms and the automatic generation of test data. Technical report, Semin. Arthr. Rheum, 1995. (Cited on pages 2 and 16.)

[Rop97] Marc Roper. Computer-aided software testing using genetic algorithms. In *Proceedings of the* 10th *International Software Quality Week (QW '97)*, San Francisco, USA, 1997. (Cited on page 21.)

[Rux06] Graeme D Ruxton. The unequal variance t-test is an underused alternative to Student's t-test and the Mann-Whitney U test. *Behavioral Ecology*, 17(4):1045–2249;1465–7279, Jul 2006. (Cited on page 78.)

[SA06] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Cited on page 32.)

[SA08] Anastasis A. Sofokleous and Andreas S. Andreou. Automatic, Evolutionary Test Data Generation for Dynamic Software Testing. *Journal of Systems and Software*, 81(11):1883–1898, 2008. (Cited on page 16.)

[Sag07] Ramón Sagarna. *An Optimization Approach for Software Test Data Generation: Applications of Estimation of Distribution Algorithms and Scatter Search.* PhD thesis, University of the Basque Country, San Sebastian, Spain, January 2007. (Cited on page 17.)

[SaIR] The Software-artifact Infrastructure Repository. `http://sir.unl.edu/portal/index.html`. (Cited on page 142.)

[SAY07] Ramón Sagarna, Andrea Arcuri, and Xin Yao. Estimation of Distribution Algorithms for Testing Object Oriented Software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pages 438–444, Singapore, 25-28 September 2007. IEEE. (Cited on page 17.)

[SB92] Nicol N. Schraudolph and Richard K. Belew. Dynamic parameter encoding for genetic algorithms. *Machine Learning*, pages 9–21, 1992. (Cited on page 18.)

[SBW01] Harmen Sthamer, André Baresel, and Joachim Wegener. Evolutionary Testing of Embedded Systems. In *Proceedings of the 14th International Internet & Software Quality Week (QW '01)*, pages 1–34, San Francisco, California, USA, 29 May-1 June 2001. (Cited on page 16.)

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005. (Cited on pages 2, 3, 27, 28, 29, 31, 35, 37, 50, 73, 87, 88 and 90.)

[Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In J. D. Schaffer, editor, *Proceeding of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1989. (Cited on page 19.)

[Sys91] Gilbert Syswerda. A study of reproduction in generational and steady state genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991. (Cited on page 20.)

[TCMM00] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000. (Cited on page 22.)

[TdH08] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008. (Cited on pages 3, 27, 31, 35, 37, 87 and 107.)

[The91] The Institute of Electrical and Electronics Engineers Inc. Standard glossary of software engineering terminology (ANSI), 1991. (Cited on page 25.)

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 119–128, Boston, USA, 2004. ACM Press. (Cited on pages 2, 17, 32 and 90.)

[Tra00] Nigel Tracey. *A Search-Based Automated Test-Data Generation Framework For Safety-Critical Software*. PhD thesis, University of York, 2000. (Cited on pages xvii, 22 and 23.)

[TS06] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. Technical Report MSR-TR-2005-153, Microsoft Research (MSR), March 2006. (Cited on page 26.)

[VIM] VIM. `http://www.vim.org/`. (Cited on page 107.)

[VM95] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995. (Cited on page 112.)

[Wap08] Stefan Wappler. *Automatic Generation of Object-Oriented Unit Tests using Genetic Programming*. PhD thesis, Technical University of Berlin, January 2008. (Cited on page 17.)

[Wat95] Alison Watkins. A tool for the automatic generation of test data using genetic algorithms. In *Proceedings of the Software Quality Conference '95*, pages 300–309, Dundee, Great Britian, 1995. (Cited on page 21.)

[WB04] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *GECCO 04: Proceedings of the 6th Annual Conference on Genetic and Evolutionary Computation*, pages 1400–1412, 2004. (Cited on page 2.)

[WBP02] Joachim Wegener, Kerstin Buhr, and Hartmut Pohlheim. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1233–1240, New York, 9-13 July 2002. Morgan Kaufmann Publishers. (Cited on page 24.)

[WBS01] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary Test Environment for Automatic Structural Testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, December 2001. (Cited on pages 2, 22, 24, 37, 87, 90, 108, 117, 123 and 145.)

[WBW07] Stefan Wappler, André Baresel, and Joachim Wegener. Improving evolutionary testing in the presence of function–assigned flags. In *Testing: Academic & Industrial Conference, Practice And Research Techniques (TAIC PART07)*, pages 23–28. IEEE Computer Society Press, sept 2007. (Cited on pages 104 and 122.)

[WBZ+08] Yan Wang, Zhiwen Bai, Miao Zhang, Wen Du, Ying Qin, and Xiyang Liu. Fitness calculation approach for the switch-case construct in evolutionary testing. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO '08)*, pages 1767–1774, Atlanta, GA, USA, 12-16 July 2008. ACM. (Cited on page 16.)

[Wei79] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979. (Cited on page 139.)

[Whi89] Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation. In *Proc. of the Third Int. Conf. on Genetic Algorithms*, pages 116–121, San Mateo, CA, 1989. Morgan Kaufmann. (Cited on page 145.)

[WM01] Joachim Wegener and Frank Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, 2001. (Cited on page 2.)

[WMMR05] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing - EDCC-5, 5th European Dependable Computing Conference, Budapest, Hungary, April 20-22, 2005, Proceedings*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005. (Cited on page 90.)

[Wri90] Alden H. Wright. Genetic algorithms for real parameter qptimization. In *FOGA*, pages 205–218, 1990. (Cited on page 18.)

[WSKR06] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 1 – 12, Portland, Maine, USA., 2006. ACM Press. (Cited on page 2.)

[WWW07] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying Particle Swarm Optimization to Software Testing. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, pages 1121–1128, London, England, 7-11 July 2007. ACM. (Cited on page 17.)

[XES+92] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of Genetic Algorithms to Software Testing. In *Proceedings of the 5th International Conference on Software Engineering and Applications*, pages 625–636, Toulouse, France, 7-11 December 1992. (Cited on page 14.)

[XXN+05] Xiaoyuan Xie, Baowen Xu, Changhai Nie, Liang Shi, and Lei Xu. Configuration Strategies for Evolutionary Testing. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC '05)*, pages 13–14, Edinburgh, UK, 26-28 July 2005. IEEE Computer Society. (Cited on page 16.)

[YH07] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA '07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 140–150, New York, NY, USA, 2007. ACM. (Cited on page 2.)