

1st Year DPhil Report

Nigel J. Tracey

30th June 1997

Abstract

This report is submitted as a first year qualifying dissertation, due 30 June 1997.

Software testing is an expensive process which few programmers like. Previous attempts to automate test-data generation are surveyed and shown to lack general applicability. For such automation methods to be widely used they must be general – both in terms of the programs and the testing problems for which they can generate test-data. A review of three modern heuristic optimisation techniques (simulated annealing, tabu-search and genetic algorithms) is given. The flexibility of these techniques means that their use may allow a general frame-work for test-data automation to be produced. A plan is given for further research which aims to investigate the use of these optimisation techniques in automated testing (for a wide variety of both functional and non-functional testing problems). The preliminary results, presented at the end of this report, show how simulated annealing can be used to find test-data for a worst-case execution time and specification falsification problems.

Contents

I	Introduction	4
1	Introduction	5
II	Field Survey and Review	7
2	Testing Functional Properties	8
2.1	Introduction	8
2.2	Automated Structural Testing	8
2.2.1	Static Test-Data Generation Methods	9
2.2.2	Dynamic Test-Data Generation Methods	13
2.2.3	Combined Test-Data Generation Methods	14
2.2.4	Optimisation Based Approach	15
2.2.5	Conclusions	15
2.3	Automated Functional Testing	16
2.3.1	Conclusion	17
3	Testing Non-Functional Properties	18
3.1	Introduction	18
3.2	Timing Analysis	18
3.2.1	Syntactic Timing Analysis	19
3.2.2	Semantic Timing Analysis	19
3.3	Resource Usage	19
3.4	Environmental Testing	20
3.5	Conclusions	21
4	Heuristic Optimisation Techniques	22
4.1	Introduction	22
4.2	Neighbourhood Search	23
4.2.1	Example	24
4.3	Simulated Annealing	25
4.3.1	Generic Implementation Decisions	26
4.3.2	Problem Specific Decisions	28
4.3.3	Example	28
4.3.4	Enhancements	28
4.4	Tabu Search	29
4.4.1	Short-Term Memory – Recency	30
4.4.2	Long-Term Memory – Frequency	32

4.4.3	Neighbourhood Exploration	32
4.4.4	Example	32
4.5	Genetic Algorithms	35
4.5.1	Encodings	36
4.5.2	Selection Methods	38
4.5.3	Genetic Operators	39
4.5.4	Parameters	39
4.5.5	Example	39
III	Future Work and Preliminary Results	42
5	Conclusions and Proposal	43
6	Preliminary Evaluation and Results	47
6.1	Automated Software Testing as Optimisation	47
6.1.1	Simulated Annealing	48
6.1.2	Tabu Search	48
6.1.3	Genetic Algorithms	49
6.1.4	Objective/Cost Functions	50
6.2	Feasibility Prototype Tool-set	52
6.2.1	Testing Problems Addressed by Prototype	53
6.3	Conclusion	56

Part I

Introduction

Chapter 1

Introduction

Software testing is an expensive process, typically consuming at least 50% of the total costs involved in developing software [Bei90]. Few programmers like testing and even fewer like test specification and design. It remains, however, the primary method through which confidence in software is achieved both by the development team and the customer [DO91]. Automation of testing is desirable both to reduce development costs and also to improve the quality of (or at least confidence in) software. While automation of the testing process – the maintenance and execution of test-cases – is taking hold commercially, the automation of test-data generation is significantly less advanced. It has been suggested by Ould that it is this area that is the most important aspect of testing requiring automation [Oul91].

Optimisation techniques are search techniques which aim to find minimal (or maximal) values of a particular cost (or objective) function. They are flexible and powerful, with the ability to obtain good results for many extremely difficult problems. Their flexibility is due to the fact that the search is simply directed by a cost function – very little other problem specific knowledge is required.

The aim of the work presented in this report is to investigate the use of optimisation techniques in the automation of software test-case data generation for both functional and non-functional criteria. To maintain a coherent focus, and to give criteria by which to judge success, the following hypothesis will form the central thread to the work.

”Non-linear optimisation algorithms are very useful for solving a wide range of complex problems. They can usefully be applied in the derivation of test cases for both functional and non-functional properties of computer systems, to improve the current state-of-the-art in testing. The use of non-linear optimisation techniques will allow generality, both in terms of the number of testing problems which can be addressed and the units which can be tested”

Structure of the Report

This report is divided into three parts. The first part contains an overview of the general issues, the hypothesis which will form the central thread to the research, and a *road-map* to the rest of the report.

CHAPTER 1. INTRODUCTION

Part two – “Field Survey and Review” – forms the main body of this report. This is divided into three chapters. Chapter 2 outlines systematic techniques for testing the functional properties of software. Black-box (structural) testing methods are examined first along with approaches to automate these. This is followed by examination of white-box (functional) testing methods and attempts at automation. The lack of general applicability of these approaches is shown to be the major problem. Many of the approaches can not deal with commonly used data-structures or control-flow structures and this severely limits their applicability outside of the research domain.

Chapter 3 focuses on methods for testing non-functional properties. The volume of literature and research in this field is comparatively small. With even less work to be found examining approaches to automate the generation of non-functional tests.

Chapter 4 introduces optimisation problems and the use of neighbourhood search. It then goes on to review three modern heuristic optimisation techniques – simulated annealing, tabu search and genetic algorithms. Each method is introduced in turn, along with suggested improvements and enhancements. Each method is applied to the same constraint satisfaction problem to give a comparison of the application process.

Part three – “Future Work and Preliminary Results” – chapter 5 presents conclusions and a proposal for further work. It is suggested that optimisation techniques present a general framework under which it may be possible to unify the approach to solving a large number of software testing problems both functional and non-functional. Chapter 6 follows with a presentation of some preliminary results. This begins with an examination of some methods for applying each of the optimisation methods to a software testing domain and also explains how to represent a number of software testing problems as optimisation problems. Details of a prototype tool-set – which has been used to investigate the feasibility of the ideas presented – is then given along with the results obtained. It is argued that these results give sufficient evidence to justify the continued research into the application of optimisation techniques to automate the generation of test-data.

Part II

Field Survey and Review

Chapter 2

Testing Functional Properties

2.1 Introduction

This chapter examines the approaches to automating the generation of software test-case data for functional properties. The testing of functional properties, which is concerned with ensuring that the software meets its functional specification, can be achieved using functional testing (*black-box*) or structural testing (*white-box*). In functional testing, the inputs and outputs from a software unit are compared with those expected from the functional specification of the software. In general, this is only possible when formal notations are used to capture the functional specification. Structural testing methods allow knowledge of the software unit to be used in systematically deriving the tests and measuring the test coverage (or completeness) [Som92].

This chapter first focuses on how to use structural testing methods to automate test-case data generation. This is followed by a review of approaches for producing test-case data from formal specifications.

2.2 Automated Structural Testing

A number of methods have been proposed to make the process of selecting software test-case data reasonably systematic (that is not to say that they make the process easy). Given a number of test-cases, structural testing methods allow a measure of coverage (or completeness) of testing to be defined. This, along with knowledge of the software structure, gives an indication as to what test data is needed to test the parts of the software which have not reached the desired level of coverage. The use of these systematic methods must form the basis of any good quality testing strategy [Bei90], whether the test-data be generated manually or automatically.

Path Based Testing This method of testing uses the underlying structure of the software as its basis. Many different criteria for test-set quality have been devised, such as statement, boolean, decision, multiple condition and decision

CHAPTER 2. TESTING FUNCTIONAL PROPERTIES

and loop coverage [Bei90]. This is the most widely used method for judging the completeness of testing. With path based testing test-data must be selected to exercise a particular path through the software with the aim of increasing the coverage measure. Much of the work on automating the generation of test-data has been carried out using path-based testing criteria.

Data Flow Testing The use of data flow criteria represents a testing strategy with stronger test coverage requirements than path based testing. The early work on the path selection criteria for data flow testing was carried out by Rapps and Weyuker, [RW85]. The automatic generation of test-data for data flow criteria basically resolves to the same problem as path based testing, however the path selection criteria changes to meet the data-flow objectives.

Fault Based Testing Mutation analysis is a fault based testing technique introduced by DeMillo et al. [DLS78]. It is based on using test-cases to show the absence (or presence) of specific faults [DO91]. Mutants of the original program are produced by altering simple operators or constant values. The testing goal is then to show that these mutants perform incorrectly (this is known as *killing* the mutants). The percentage of the mutants which are *killed* by a test-set is known as the *mutation score*. The large number of mutants which will be produced for any reasonably-sized software unit means mutation testing can be very time-consuming and expensive. Automation of the test-data generation process may make this approach more feasible.

The remainder of this section goes on to review the approaches taken in the literature to generate structural test-case data automatically. These approaches can be divided into two categories as shown in figure 2.1. Firstly, there are the static approaches. These methods make use of symbolic execution to statically determine path-conditions¹ and then use various means to derive test-data from them. Secondly, there are the dynamic methods. These methods execute the software under test with the aim of searching for suitable test-data. Only the work by [OP96] attempts to combine both the static and dynamic approaches.

Figure 2.1 also illustrates the lack of research during the 1980's. During this period the focus of almost all research turned to formal methods. However, since 1990, research into the software testing field has grown once more. This is probably due to the complexity and expense of applying formal methods and also because of the realisation that the use of formal methods does not obviate the need for good software testing. In fact, good software testing can make the application of formal methods a great deal more economical.

2.2.1 Static Test-Data Generation Methods

One of the earliest attempts at automating software test-case data generation was Clarke's work on symbolic execution [Cla76]. Symbolic execution systems work by traversing a control flow graph and building up symbolic representations of the internal variables in terms of the input variables. Branches within

¹Path conditions express constraints on input variables. Any data which satisfies these constraints will cause the execution of the specified path.

CHAPTER 2. TESTING FUNCTIONAL PROPERTIES

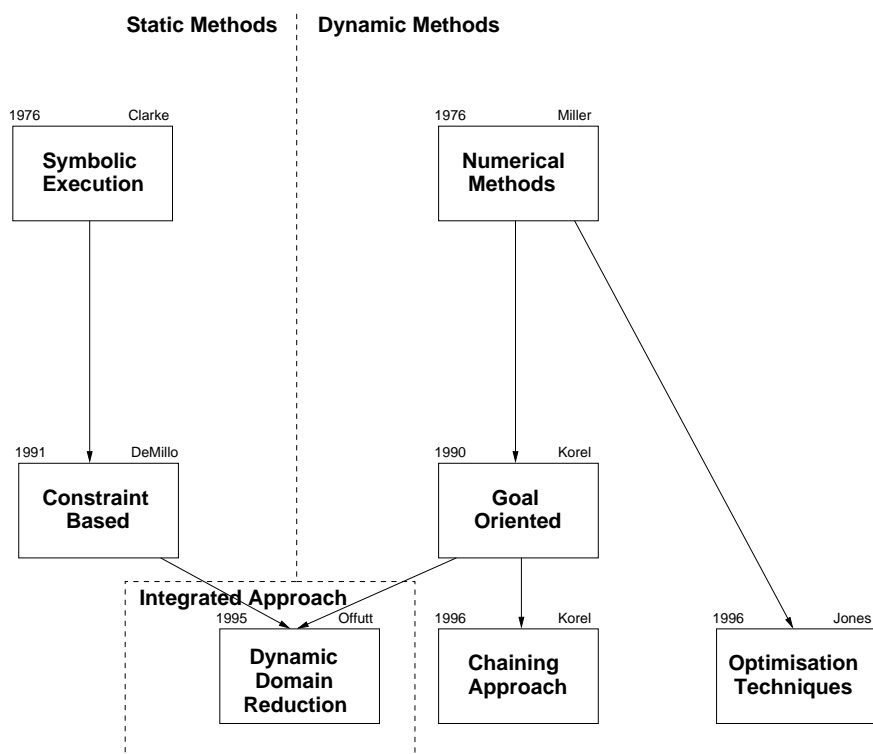


Figure 2.1: Approaches to Automate Test-Case Data Generation

the code introduce constraints on the variables. It is the solution to these constraints which represents the test-data for the path. It is at the path selection stage where the required structural testing criteria are important. For example, for statement coverage a set of paths must be selected which between them execute every statement in the software-unit. Clarke devised a method to symbolically execute a given path of an ANSI Fortran program to generate a set of constraints. If these constraints were linear then linear programming techniques were used to obtain a solution. This solution represents input data which forces execution of the specified path.

Clarke's work suffered from several limitations. In order to circumvent the path explosion problem, the developed system contained no path selection method. Instead, it required the user to completely specify the path. If array indexes depend on input data then the system could not resolve the reference and therefore use them to form constraints². The approach adopted by Clarke was simply to note each occasion when this problem occurred in order to attempt to gain some empirical evidence for the level of the problem.

Other early notable work on symbolic execution includes the work on EFFIGY [Kin76]. EFFIGY was a complete system for testing and debugging programs written in a PL/1 style language. The system aided the management of test-cases and the task of proofs of symbolic expressions. The SELECT system built on this work by King [BEL75] by allowing arbitrary symbolic assertions to the source code to aid the task of program proofs.

CASEGEN [RHC76] improved the processing of arrays. The approach described delayed the symbolic execution of arrays when it was dependent on input data until the constraint satisfaction stage. Values for variables upon which array access depends were determined first which in turn allowed the array access to be completely determined. This method required creating a new symbolic instance of the array, identical to the previous instance except for the element in question. This approach, however, can require large amounts of memory in which to store all of the symbolic representations of each array. This work also proposed an improved constraint satisfaction method.

1. Sort variables into order v_1, v_2, \dots, v_n .
2. Place constraints into sets
 - Constraints in set s_i must access variable v_i and only other variables from v_1, v_2, \dots, v_{i-1} .
3. Generate values in the order v_1, v_2, \dots, v_n .
 - Checking that the value for v_i satisfies all the constraints in s_i .
4. When no value can be found, back-track to previous variable.

Coen-Portisini *et al.* [CpD93] presented a method to help overcome some of the limitations with regards to arrays. Their approach involved binding each variable to a set of symbolic values and to the constraints under which each value would hold. These sets are incrementally updated to attempt to reduce the excessive growth of both computational space and time. Despite

²The same problems occur with the use of pointers or access types.

these improvements, weaknesses are still present regarding arrays, pointers and unbounded loops with symbolic execution. The testing of higher level software units may also make symbolic execution prohibitively expensive. This is because all subprogram calls must be symbolically executed as if they were *in-lined* at the position of the call.

The work by DeMillo *et al.* [DO91], [Off91], [DO93] developed a method known as constraint-based testing. This is a symbolic execution technique aimed at mutation analysis, however the approach is applicable to path and data-flow based testing. The central theme is the definition of a *necessary-condition*. This is a constraint that describes the necessary condition to kill a program mutant. This, however, is not enough to guarantee that a mutant will be killed. In general, generating the sufficient condition is an intractable problem. The power of the approach is enhanced by the definition of a predicate constraint. When a predicate expression is mutated, additional constraints are formed. These force the result of the mutated predicate to be different from the original. For example, if the original predicate is $X > 0$ and this is mutated to $Y > 0$, the necessary condition would be $X \neq Y$ and the predicate constraint would add $X > 0 \neq Y > 0$.

A search procedure known as *domain reduction* is used to attempt to find solutions to each of these constraint systems, formed from the path condition, the necessary condition and the predicate condition for the particular mutation. This procedure works by using information in the constraint system to reduce the domains of the variables as follows.

1. Variable domains are initially derived from type information.
2. Reduce a variable's domain using information in a constraint. This information is propagated throughout the constraint system.
 - (a) For constraints of the form $X \boxed{\text{relop}} c$, the domain of the variable X is reduced.
 - (b) For constraints of the form $X \boxed{\text{relop}} Y$, the domains of both X and Y are reduced.

Where $\boxed{\text{relop}}$ is a relational operator.
3. Continue from step 2 until no further reductions are possible.
4. Assign a random value to the variable with the smallest domain, effectively giving a domain with a single element. This is then propagated throughout the constraint system. Continue from step 2.
5. If all variables have been assigned a value then success, otherwise restart from step 1. The randomised nature of the search means that subsequent attempts may be successful.

Mutation scores have been achieved in the region of 95% to 100% using this method for several sample programs. Human mutation testing on the same programs resulted in similar scores, but in one to two orders of magnitude more time. Later research into this method by [Off91] [DO93], has focused on increasing the efficiency of the test-data set. A single test-case is likely to kill

many mutants and therefore there is scope to reduce the number of test-cases generated.

The use of symbolic execution remains the weakest link in constraint based testing: loops are only analysed in terms of their entry and exit points; only the first two array elements are analysed; the problems of pointers and dynamic memory are not addressed.

2.2.2 Dynamic Test-Data Generation Methods

The first use of dynamic methods to automatically generate test-data is presented in [MS76]. The use of dynamic methods is based on the contention that test-data generation is sometimes best formulated as a numeric maximisation (or minimisation) problem. While this has the disadvantage of being a *heuristic* method which cannot guarantee to produce test-data even when such data exists, it does have that advantage that it may represent a more general solution overcoming some of the problems of symbolic execution.

The approach requires that a straight-line version of the test program be produced for the particular path being tested. All conditionals within the path are replaced by constraints. A function is then used to map these constraints onto a real-value. The function is formulated in such a way so that it is negative when any of the constraints are not satisfied and positive only when all the constraints are satisfied. Finding test-data then involves using numerical techniques for constrained maximisation, stopping as soon as the function becomes positive.

The results from this system appeared very encouraging. Suggestions for further work included attempting to apply global heuristic optimisation techniques to improve the results of the search. However, little work has investigated further this promising approach [Inc87].

Later work by Korel [Kor90] presented a different dynamic technique known as the *goal oriented* technique. Again, the aim was to overcome the limitations of symbolic execution, this time by utilising information which is only readily available at run-time. This method executes the test unit under the control of a monitor. If an unwanted branch is taken, the monitor suspends the execution of the subprogram. A real-valued function is associated with the branch. This function is devised so that it will only be negative when the desired branch is taken. Simple function minimisation techniques are then used to attempt to find data which will execute the same path up to this branch, but which will now take the correct path.

Because this technique is dynamic, it is possible to use dynamic data-flow information in this search process. This information is expressed as an *influences* graph which shows which variables influence which branches. This allows the search to be modified so that variables which influence the rogue branch, but not preceding branches, are changed first. Again this research proposed the use of global optimisation techniques as the way forward.

Building on the goal-oriented approach, Ferguson and Korel [FK96] [Kor96] proposed the *chaining approach*. This extends the search and execution methods to include the notion of event sequences. An event sequence is a sequence of tuples $e_i = (n_i, S_i)$, such that $n_i \in$ of the set of nodes in the control-flow graph and S_i is a set of variables known as the *constraint set* which must not be modified between the nodes n_i and the node from the next tuple in the sequence.

Each time an unsuccessful branch is traversed, the execution is stopped and the function minimisation search initiated. If the search successfully finds new input values which correct the execution path, execution proceeds. However, if the search process fails, the chaining approach generates a number of event sequences. One event sequence is generated containing tuples stating that each last-definition node of all variables used at the problem node must be executed en-route to the problem node, and also that the path taken from the last-definition node to the problem node must be a definition clear path with respect to the variable in question. This is clarified with a simple example.

- Given an initial event sequence, $\langle (start, \emptyset), (end, \emptyset) \rangle$.
 - This states that the initial goal is to find a path from the *start* node to the *end* node, which can modify any variable on the way.
- Assume that at node *p* a branch is taken which cannot lead to the *end* node, *p* becomes the problem node.
- The chaining approach first uses a search procedure utilising function minimisation techniques to correct the control flow at node *p*.
- If the search fails, a number of new event sequences are generated. Assuming the node *p* only uses the variable *X* and that nodes *a* and *b* modify the value of *X*.
 - $\langle (start, \emptyset), (a, X), (end, \emptyset) \rangle$.
 - $\langle (start, \emptyset), (b, X), (end, \emptyset) \rangle$.
 - This states that the control flow should start at node *start* and progress (via any path, modifying any variable) to node *a* (or *b* in the second event sequence). Then from node *a* (or *b*) the control flow should progress to node *end* along a definition clear path with respect to variable *X*.

Using this method, a tree of event sequences is generated which is then searched in a depth-bounded depth-first manner. It can be seen that the goal-oriented approach is equivalent to bounding the depth at the root of the event sequence tree. As the authors point out, this approach to generating test-data is not optimal. Instead it represents a way forward for future research. Again, the use of global optimisation techniques may improve the result. Also, the concept of program slicing may reduce the computational expense involved in repeated execution of the test unit.

2.2.3 Combined Test-Data Generation Methods

The only approach which attempts to combine both a static and dynamic approach is the dynamic domain reduction procedure (DDR) [OP96]. This builds on both the constraint-based [DO91] and goal-oriented [Kor90] methods. The dynamic nature of the method aims to give improved results, particularly in the presence of arrays.

The DDR procedure walks the control flow graph of the test unit. All variables start with domains as assigned by their type information. As the edges

of the path are traversed, their contents are symbolically executed. When a decision node is encountered the variable domains are reduced according to the branch predicate so that condition will always be true. A binary search of the possible reduction points is used through back-tracking to help correct any wrong decisions made.

For example, if two variables have domains $X : [1..10]$, $Y : [5..20]$ and a branch in the control flow graph is traversed containing the branch predicate $X < Y$, the domains are altered by DDR to $X : [1..7]$, $Y : [8..20]$. The branch predicate will always be true for any pair of values from these new domains.

The work contains some proof-of-concept evaluation of the new procedure. This showed some good results with between 93% and 100% all-uses coverage. However, in the presence of arrays the method appeared to perform little better than the earlier constraint-based method.

2.2.4 Optimisation Based Approach

Despite the suggestion of this approach back in the 1970s [MS76] very little work has been done. Within the last year the first published work in this area has appeared. The work by Jones *et al.* [JSE96] uses genetic algorithms (see section 4.5) to generate structural test-data. The use of genetic algorithms allows complex data to be generated, as during the generation process simple bit-strings are used. If particular data causes the execution of a sibling of the desired node then the fitness function value is related to the branch predicate. In this way the genetic search will guide the data towards the correct node.

This work has been shown to produce 100% statement coverage in far fewer test-cases than simple random testing. Also, when tested against mutation criteria the system has achieved mutation scores of up to 99%, out-performing the constraint-based method. However, this approach to applying genetic algorithms is too simple. The genetic operators of mutation and crossover can result in illegal bit patterns for the given type. There is no presentation of how this problem could be addressed or how one might devise more intelligent genetic operators to avoid this problem.

2.2.5 Conclusions

Most of the work on test-data generators has been based on symbolic execution [BEL75], [Cla76], [DO91]. The weaknesses of symbolic execution – analysis of loops, recursion, array indices which are data dependent and pointers – have been well researched. No ideal solution has yet been found. Even if these problems could be addressed effectively, there still remains the issue of how to solve the constraints. In general, the constraint satisfaction problem is intractable. However, more research is needed to discover whether constraints produced from *real* programs fall into a sub-class which can, in fact, be solved efficiently.

Dynamic approaches to generating test-case data have been shown to perform effectively for a number of testing-criteria. The ability to cope well with arrays and dynamic memory usage allows them to generate test-data for a wider class of software. However, less research has been carried out to test these results on a large scale. There is a real need for some good quality empirical results. Many of the approaches have simply been demonstrated on a handful of sample programs. Also, research into the application of global optimisation

approaches has been slow to take-off. Since being recognised as a possible way forward in the 1970's no work has been carried out until very recently.

Global optimisation techniques would appear to represent a very general solution to the test-case data generation problem. The ease with which the work by Jones *et al.* was turned from statement coverage to mutation analysis is testament to that.

2.3 Automated Functional Testing

Functional testing, as stated earlier, treats the software under test as a *black-box*. Random test generators are a degenerated form of black-box testing as they do not use the source code to generate tests. Random testing is the method by which all other test-case data generation methods can be judged [Inc87]. As a sole form of testing it is dangerous as it promotes confidence that software has been tested thoroughly. Random testing tends to produce large numbers of test-cases to achieve the same test quality metrics as alternative testing methods. In his book, Beizer [Bei90] points out that random testing is intuitively the worst possible way of testing software. However, if applied as a supplementary method of testing, especially at the system level, random testing can be a very effective method of stress testing when compared with tests generated by other methods.

Apart from random testing, perhaps the earliest form of automated black-box testing was syntax based testing. This approach selected test-data using a formal grammar of the input language. The structure of the program under test was unimportant – all that was required was the formal syntax of the input to the system. Thus the method was really only applicable to systems where an input grammar had to be derived (such as compilers, command-line systems, etc . . .). A simple algorithm was presented in [Bur67] which allowed sentences of a language to be generated from the grammar of the language itself. In effect this is the compliment to a parser.

An alternative form of functional testing is to use the specification to generate the tests. In order for automation to be possible, the specification must be sufficiently formal. Testing from formal specifications has a number of advantages – it has the potential to detect missing functionality, it is independent of implementation language and it provides scope for an automated oracle³. Perhaps the most promising work in this area is the research into testing from already established specification languages used in other areas of software engineering. The alternative is to write specific specifications for testing purposes. This introduces a large burden on the tester and increases the risks of inconsistencies between the different representations of the system's requirements.

Functional testing from formal specifications is comparatively advanced for telecommunications (or protocol based) systems. The specification of such systems is often given in the form of a finite state machine (FSM). Many test selection methods have been developed for such specifications – the transition tour, W-method, distinguishing sequence method, unique-input-output method and the partial W-method [FvBK⁺91, Bur96]. However, for more general classes of software, finite state machines are not such a suitable way of expressing the

³“Any (often automated) means that provides information about the (correct) expected behaviour of a component. Specifies the expected outcome for a specified input.” [Bei90].

CHAPTER 2. TESTING FUNCTIONAL PROPERTIES

specification. For these, software systems specification languages such as VDM-SL and Z are more suitable. The work by Yang [Yan95] is one approach to generating test-data directly from a Z specification. This work uses genetic algorithms to search for test-data using the Z specification. From the declaration part of the Z, the specification type and domain of each variable can be derived. A tree of routes (or paths) describing execution flow is obtained by analysis of the Z. The genetic algorithm is then used to search for test data which are both on and either side of the boundary specified by each expression in the route. For example, an expression $exp1 > exp2$ will cause a search for test-data $exp1 = succ(exp2)$ (valid data test-case), $exp1 = exp2$ (invalid data test-case) and $exp1 = pred(exp2)$ (invalid data test-case). The work points to the need further analysis of Z specifications to allow more coverage of Z functions, investigation into other search methods such as simulated annealing and tabu search, and measurement of test effectiveness to gain good empirical evidence.

2.3.1 Conclusion

While the idea of testing from high-level specifications is certainly not new – Hayes [Hay86] in 1986 presented a manual approach to deriving test-cases from VDM-SL specifications – research into automating functional testing is still in its infancy. Further research in this area is needed. It would represent a great advance in software engineering if an efficient way to test software directly from a high level specification could be found.

Chapter 3

Testing Non-Functional Properties

3.1 Introduction

This chapter examines the approaches used to validate non-functional properties of software. Testing non-functional properties is concerned with ensuring that the constraints imposed by process, product or external requirements are met [Som92]. Research into testing non-functional requirements is limited, with very little work on automation of the testing process. The majority of research into non-functional requirements has focused on analysis techniques. This section focuses on the major non-functional requirements for high-integrity (and real-time) software and the methods used to validate these requirements.

3.2 Timing Analysis

Timing analysis is perhaps the most basic non-functional property for safety-critical real-time systems. There are two distinct levels of timing properties:

- Worst Case Execution Time (WCET) for sequential code is analysed at the unit level;
- Worst Case Response Time (WCRT) is analysed at the system level.

In order to analyse the WCRT of a system it is a pre-requisite that the WCET of all units is known. The focus for this discussion will therefore be the validation of WCET. The majority of the research into obtaining WCETs has focused on static analysis techniques.

The static analysis approach requires a low-level mapping of statements to execution time. Obviously, such a mapping requires detailed knowledge of both the compiler and the target hardware. Because of the complexities modern hardware the mapping is usually pessimistic. For example, compiler optimisations, cache memory and pipelining all make it difficult to build a deterministic model of the hardware. The static analysis approaches for obtaining timing information for software can be divided into two categories: syntactic analysis and semantic analysis.

3.2.1 Syntactic Timing Analysis

This approach simply obtains the longest path through the code using pure syntactic analysis. Instruction execution times are simply summed for all instructions on each path. No path feasibility analysis is performed. This means that the WCET obtained may be for a path which it is not possible to execute. While this approach has low overheads it produces very pessimistic WCET.

3.2.2 Semantic Timing Analysis

This approach incorporates path feasibility analysis in an attempt to remove infeasible paths from the results. Two approaches can be used to incorporate path feasibility analysis. Firstly, explicit annotations can be supplied by the user [Par93] [CBW96]. This approach places a large overhead on the user as both the derivation and proof of the annotations can be extremely complex. While annotations can work well for conditional statements, the subtleties of loops can be ignored. Thus while annotations can remove many infeasible paths, they only perform partial path feasibility analysis. The second approach to performing feasibility analysis is to use symbolic execution and a constraint-solver. The symbolic executor generates path traversal conditions which if satisfiable indicate that the path is feasible. This approach requires (i) that the symbolic executor can deal with the language constructs being used and generate path conditions and (ii) that the solver is complete with regard to the conditions generated. If these two conditions can be met then total path feasibility analysis can be performed. This would mean that the WCET obtained would always be for a feasible path. However, pessimism would still remain due to the low-level model of the hardware. The language subsets in which most high integrity software are programmed, and the advances being made in constraint solving mean that meeting these conditions may be feasible.

3.3 Resource Usage

Resource usage such as stack-use or memory use can be handled in the same framework as for timing analysis. The low-level model simply has to be altered to map instructions on to their stack or memory use.

Automation of the testing of resource allocation mechanisms for telecommunications software has been the focus of recent work. The work by Averitzer [AW95] presents a number of slightly different algorithms for generating load-test suites. The telecommunications focus is due to the requirement that the software operational profile is modelable using Markov chains. Markov chains can accurately model the operation of telecommunication software, however it may also be possible to model other (particularly safety critical real-time software) using Markov chains. The algorithms presented generate one or more test-cases per Markov state. Due to the state explosion problem, even for relatively small systems, it is necessary to select only a subset of Markov states when generating test-cases. This is achieved by restricting the analysis to those states which are most likely to be executed. The three algorithms presented in [AW95] are summarised below.

1. *Deterministic State Testing (DST)* – This algorithm generates only one test-case per significant Markov state. This assumes that the execution of the state which will uncover any errors and that the route to the state is unimportant.
2. *DST With State Transition Checking* – This extends DST by also considering each transition into a Markov state as significant. This allows more than one test-case to be generated per Markov state.
3. *DST With Length N Cycles* – This extends DST still further. This algorithm considers sub-paths leading to Markov states rather than just the single transition. This generates more test-cases but has been shown to be effective at detecting resource leaks.

More research is needed into resource-use testing for software outside of the telecommunication domain, particularly for software which cannot be modelled by Markov chains.

3.4 Environmental Testing

Environmental bugs are caused by limitations in the execution environment of software. A number of such restrictions exist in any execution environment. The following gives some examples:

1. *Numeric* – This includes limitations in the machine’s representation of numeric values, both size and precision. Further restrictions may be imposed by the language in the form of strict bounds on variables’ values.
2. *Memory* – There will never be an infinite amount of memory available.
3. *System* – Processors may contain bugs, compilers may produce incorrect code, and operating systems may not execute the code correctly.

The work presented by Spafford [Spa90] introduces an extension to mutation testing in order to address some of these issues. This work details an extension to allow limitations in integer value representation to be uncovered. The basic approach involves generating mutants which can only be killed by test-cases which cause overflow or underflow in a particular integer expression. This is achieved using two new mutations - one for overflow and one for underflow. The effect of the mutation is to replace an integer expression with a function whose parameter is the same integer expression. This function returns the same value as the integer expression whenever overflow (or underflow) does not occur. However, if overflow (or underflow) does occur the function flags that the current test-case has killed the mutant.

Such environmental testing may be able to find classes of errors which current testing strategies do not address efficiently. However more research is needed into the possibilities of producing complete sets of environment specific mutants. One useful direction for further research presents itself. The automation of test-case generation for bounds checking and exception conditions may greatly increase the efficiency with which confidence of exception free execution can be obtained.

3.5 Conclusions

Relatively little research has been carried out in the field of testing to validate non-functional properties. In practice it seems that an *ad hoc* approach of reusing functional tests to perform the non-functional testing (such as execution time-testing) is used. This approach is little better than blind random testing. Obviously given the lack of research into testing for non-functional properties there is an even greater lack of research into automation of the process.

Chapter 4

Heuristic Optimisation Techniques

4.1 Introduction

This chapter covers the background information required before discussing the application of optimisation techniques to software testing. There are several methods which have ‘grown-up’ over the last decade or so. A brief review of their common history is followed by a more detailed examination of each technique. To illustrate and compare the use of each technique an example is given — this shows how each method can be used to solve a simple constraint satisfaction problem.

Finding optimal solutions to problems which can be structured as a function of a number of *decision* variables, and possibly a number of *constraints* has occupied researchers for the past half-century. A more formal description of such problems can be formulated as shown in below:

$$\begin{array}{ll} \text{Minimise} & f(\mathbf{x}) \\ \text{Subject to} & g_i(\mathbf{x}) \end{array}$$

\mathbf{x} , represents a vector of decision variables and $f()$ represents functions such that if \mathbf{X} represents all feasible solutions then $f : \mathbf{X} \leftrightarrow \mathfrak{R}$. Hence it can be seen that the object is to find a $\mathbf{x} \in \mathbf{X}$ which minimises $f()$ while satisfying $g()$. Maximisation problems can be expressed in the same manner with the obvious modifications.

Such optimisation problems occur commonly. Famous examples include the knapsack problem, PCB routing and the travelling salesman problem. A naive approach to solving such problems is to enumerate all feasible solutions. Given this enumeration the value of $f(\mathbf{x})$, the objective function¹, can be calculated for all \mathbf{x} . The solution is then simply the value of \mathbf{x} which gives the best value of the objective function. This naive approach is obviously an inefficient method. While in principle it is possible to solve any problem using this naive method, in practice the enormous number of feasible solutions for any reasonably

¹The objective function (also known as the cost function) gives a quantitative measure of the *goodness* of a solution (or assignment of values to decision variables).

sized problem means the approach is computationally intractable. This can be demonstrated with a simple example. If a machine can enumerate all solutions for a 20 city travelling salesman problem (TSP) in 1 hour then a 25 city problem will take approximately 7.3 centuries [RSORS96]. One further problem with the enumeration approach is that it becomes impossible if the decision variables are allowed to take on continuous rather than discrete values.

Traditional approaches to solving optimisation problems improve on the naive enumeration approach. Such methods include linear programming, integer programming and calculus-based methods. These approaches all place restrictions on the objective function or the constraints. For example calculus-based methods require differentiability. It has been shown that for most problems the computational effort required by these approaches grows exponentially in the problem size [Ree96]. So with regards to practicality these approaches are no better than the naive enumeration approach (i.e. for a reasonable sized instance of a problem they are prohibitively expensive). Indeed is it possible to show that many interesting optimisation problems are intractable, that is their decision versions are *NP complete* [Saa91], thus it is *unlikely* that there are any tractable solutions.

The research focus has therefore moved towards heuristic methods. These aim to build a more realistic model of the problem (i.e. with less restrictions on problem formulation) which has a high chance of obtaining a ‘good’ solution. The heuristic approach can be defined as in [Ree93]:

A heuristic technique is a method which seeks good (i.e. near optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is.

4.2 Neighbourhood Search

A feature of most optimisation problems is the presence of many more local optima than global optima. The concept of local optima requires the introduction of idea of a *neighbourhood*. Informally this is a set of solutions² which are *close* in some respect to a given solution. These can be defined more formally as follows. For each solution $\mathbf{x} \in \mathbf{X}$ there is a set of neighbouring solutions, $N(\mathbf{x}) \subset \mathbf{X}$, this is called the *neighbourhood* of \mathbf{x} . It is also the case that $\forall \mathbf{y} \in N(\mathbf{x})$ that \mathbf{y} can be reached by the application of an operation called *move* on \mathbf{x} . A local optimum \mathbf{y} with respect to the neighbourhood of \mathbf{x} can then be defined as ³.

$$\exists \mathbf{y} \in N(\mathbf{x}) \cdot \forall \mathbf{z} \in N(\mathbf{x}) \cdot f(\mathbf{y}) \leq f(\mathbf{z})$$

This states that the local optimum \mathbf{y} yields the best (in this case minimal) value of the objective function when compared with every other solution in $N(\mathbf{x})$.

²Solution, here, refers to an assignment of values to the decision variables of \mathbf{x} .

³This definition is for minimisation problems, a simple modification is necessary to define local optima for maximisation problems

Neighbourhood search is a heuristic method which aims to overcome the intractability of solution enumeration or *exact* solution approaches. This is achieved by only searching a small subset of the total number of feasible solutions. There are two slightly different methods of neighbourhood search. From an initial solution, \mathbf{x} , the *steepest descent* method finds the best solution in $N(\mathbf{x})$. If this is a better solution than \mathbf{x} then the process starts again from this new solution. In *random descent* instead of finding the best solution in $N(\mathbf{x})$, random samples are taken until a solution which is better than \mathbf{x} is found. Both methods terminate when there is no solution better than \mathbf{x} in $N(\mathbf{x})$.

Neighbourhood search is attractive for a number of optimisation problems as they have natural neighbourhoods. For example, if the number of elements in the solution are known then the neighbourhood structure contains those solutions which can be obtained by swapping a given number of elements in and out of the current solution. This *k-swapping* neighbourhood structure has been applied to many of the traditional optimisation problems such as the travelling salesman problem.

There is an obvious trade-off between the quality of the result and the computation time. Smaller neighbourhoods can be searched much more efficiently. However, they are more likely to converge to a local optimum and therefore they will normally give a worse approximation to the global optimum. Larger neighbourhoods will slow down the search process, but will also reduce the number of local optima⁴ therefore increasing the probability of finding the global optimum.

The failing of neighbourhood search to give good approximations to the global optimum has been widely researched [RSORS96]. For example, one weakness of neighbourhood search is that the result is dependent on the initial solution. Many of the more modern heuristic optimisation techniques are based on neighbourhood search, but each uses a slightly different approach to attempt to reduce the propensity to find only locally optimal solutions.

4.2.1 Example

To illustrate the use of neighbourhood search this example shows an application of the method to solving a simple constraint. The constraint is as follows :-

$$(x \Leftrightarrow 2) \times (x \Leftrightarrow 12) \times (x \Leftrightarrow 16) \times (x \Leftrightarrow 30) < \Leftrightarrow 10000 \wedge x \geq 0 \wedge x \leq 31$$

For the purposes of this example the neighbourhood will consist of the solutions within ± 3 of the current value for x . As can be seen from figure 4.1 this neighbourhood gives a local optimum when $x = 5$, while the global optimum (and a solution to the constraint) is at $x = 25$.

Figure 4.1a shows the progress of the neighbourhood search from an initial random solution S_0 using the steepest descent method (although random descent would follow a very similar pattern). It can be seen that the search terminates with $x = 25$ which is both the globally optimal solution and a solution to the constraint.

⁴This is because there can only be local optima with respects to the neighbourhoods, therefore if these are bigger there will be fewer of them.

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

However, figure 4.1b illustrates the neighbourhood search's dependence on the initial solution. This time the initial solution S_0 is where $x = 12$. As the search proceeds neighbourhood search is *pulled* down into the local optimum at $x = 5$, from which it cannot escape. Therefore from this initial solution neighbourhood search fails to find a solution to the constraint as it terminates with only a locally optimal solution which is not less than 10,000.

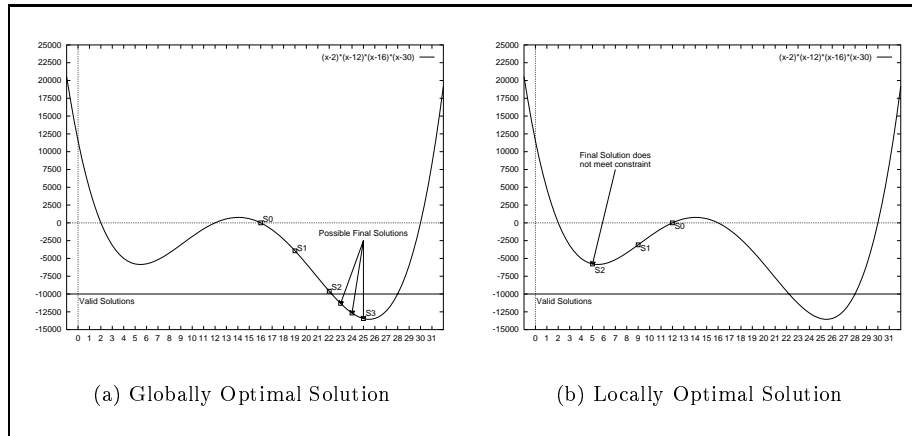


Figure 4.1: Neighbourhood Search Example

4.3 Simulated Annealing

The ideas which formed the basis for simulated annealing were first published in 1953 [MRTT53]. This work presented an algorithm to simulate the cooling of a material using a heat bath (a physical process known as *annealing*). In annealing the objective is to cool the material slowly so that a *near* perfect lattice crystal structure (i.e. a state with minimum energy) is obtained. Thirty years later Kirkpatrick *et al.* [KCDGV83] suggested that a form of simulated annealing could be used to solve complex optimisation problems such as those found in design, placement and wiring of complex electronic systems. This work suggested a close analogy with the physical annealing process.

Simulated Annealing		Physical Annealing
Feasible Solutions	\Leftrightarrow	States of the physical system
Quality of Solution	\Leftrightarrow	Energy of a given state
Control Parameter	\Leftrightarrow	Temperature

This analogy allows simulated annealing to be used to find solutions to optimisation problems which are *near* optimal solutions.

The process of simulated annealing is similar to the random descent method of neighbourhood search in that a neighbourhood is randomly sampled. The process attempts to reduce the problems of simple neighbourhood search and avoid local optima by allowing up-hill (or inferior) solutions in a controlled manner. The idea behind this is that it is better to accept a short-term penalty

in the hope of finding significant rewards longer-term. In accepting an inferior solution the search aims to escape from locally optimal solutions in order to find a better approximation to the global optimum; the control parameter (or temperature) is used to control the acceptance of inferior solutions. Acceptance of worse solutions depends upon the the degree of inferiority and the current temperature. An outline of the simulated annealing search process is shown in figure 4.2

Simulated annealing is a general purpose search strategy. Much of the algorithm remains completely unchanged across problem domains. [Dow93] classifies the implementation decisions which must be made into two categories. These are the generic decisions and the problem specific decisions.

4.3.1 Generic Implementation Decisions

These generic decisions are parameters of the annealing algorithm itself and are not specific to the problem being solved. They include the *cooling schedule* which defines a finite sequence of values for the temperature. There is also the related decision of how many solutions to sample at each temperature value. In order that the final decision be independent from the starting point⁵ the initial temperature must be *hot* enough. This means that almost any solution will be accepted initially allowing free movement around the solution space. This is often achieved by starting with a fixed temperature and increasing it until a given proportion of solutions are accepted.

The *cooling* of the temperature parameter is central to the success of the search process. There is basically a choice between a large number of iterations at a small number of temperatures or a small number of iterations at a larger number of temperatures. Generally one of two cooling schemes (or a slight variation of them) is used. The first of these is a geometric reduction of the temperature by multiplication by a constant, α . The empirical evidence suggests values of between 0.8 and 0.99 for α as being the most successful. The number of iterations at each temperature in this scheme is usually increased geometrically or arithmetically. This allows greater search at lower temperatures to ensure that local optima have been fully explored. It is desirable to accept at least some solutions at each temperature (to ensure that the neighbourhoods are searched sufficiently). An alternative to geometric increases in iterations is therefore to use feedback from the annealing process. In this way the time spent at large temperatures will be small and the time spent at small temperatures will be large. Once the temperature gets small the number of accepted solutions may get so small that an infeasible number of iterations are required to accept the desired number of solutions, hence a maximum limit is normally also imposed. The second common cooling schedule is that first proposed in [LM86], where only one iteration is performed at each temperature. However, the temperature is reduced extremely slowly. Both [Dow93] and [vLA88] give details of many other cooling schemes with reference to theoretical results of convergence to optimal solutions.

⁵Recall that a dependencies between the starting solution and the result in neighbourhood search was one of its weaknesses.

4.3.2 Problem Specific Decisions

The decisions specific to the problem include the representation of the solution space, the neighbourhood structure and how to quantify the cost (*goodness*) of a solution. Obviously, it is not possible to define a series of rules as to how to make these decisions. There are, however, a number of guidelines which have arisen from statistical analysis and empirical evidence of how simulated annealing works. Much of the work on proof of convergence to globally optimal solutions requires that every solution should be *reachable* from every other. This places constraints on both the representation of the solution space and the neighbourhood structure. Further constraints are placed on the neighbourhood structure by Hajek's theorem that it is best to avoid neighbourhoods which give a spiky or flat cost surface.

4.3.3 Example

Figure 4.3 shows the progress of a simulated annealing search for a solution to the same constraint problem as in section 4.2.1 with the same definition of the neighbourhood. From the initial solution S_0 (note that this initial solution caused neighbourhood search to fail) the search wanders freely around the solution space due to the initial high temperature. As the temperature is reduced it is still possible for the search to climb out of the local optimum. The search can progress from S_a to S_b to S_c as these worse solutions will still be accepted with a probability based on the current value of the temperature. During the later stages of the search, when the temperature has been reduced still further, the inferior moves will have a remote probability of being accepted. This allows the search to progress from solution S_c to S_d to S_e to S_f . At solution S_f the search will terminate due to the stop criteria which states that a termination should occur as soon a solution which satisfies the constraint is found.

4.3.4 Enhancements

Johnson *et al.* [JAMS89] discovered that the calculation of the exponential acceptance criterion accounted for nearly a third of the total computation time. They suggested two alternatives to improve on this situation. The first involves approximating the calculation of the exponential with $1 \leftrightarrow \frac{C_Cost - N_Cost}{Temperature}$ (where C_Cost and N_Cost represent the costs of the current and new solutions respectively). The second approach requires a lookup table to give a discrete approximation to the exponential calculation. Johnson's work showed that this reduced the computation time by nearly one-third with no adverse effect on the quality of the final solution.

Tovey [Tov88] suggested that improvements can be made to the neighbourhood structure by using a penalty function to enforce constraints on the solution that improvements can be made to the neighbourhood structure. This work suggested that it is more efficient to restrict the neighbourhood to changes which will affect the result of the penalty function. More recent empirical evidence suggests that the most efficient strategy is to use this restricted neighbourhood with a fixed probability with the normal neighbourhood being used for the remaining iterations.

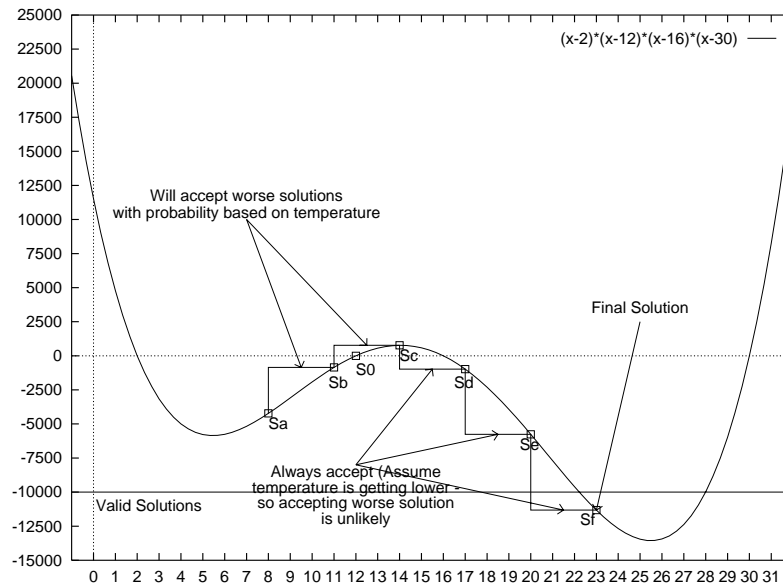


Figure 4.3: Simulated Annealing Example

Another enhancement which has been suggested [Ree96] is to perform some pre-processing in order to find a starting position. The problem, however, is that the starting high temperature effectively ignores this good starting point. It is therefore necessary to start with a much lower temperature. Care must be taken to ensure that the temperature is not set too low. The combination of a good starting solution and a low starting temperature may mean that the search gets stuck at the local optima represented by this starting good solution.

4.4 Tabu Search

Tabu Search like simulated annealing is based upon the idea of neighbourhood search, but with differing approaches to avoid local optima. Whereas simulated annealing is a stochastic technique based upon randomised search, the standard implementations of tabu search are deterministic [Bat96]. Again it is the modelling of a natural process which forms the basic ideas of the method. In this case the aim is to model the intelligent use of memory which humans use in a search process. This model of intelligent memory is used in two ways. Firstly, it is used to attempt to avoid repeated search of the same area of the solution space. This is achieved by maintaining a *tabu-list* of neighbourhood members which are *tabu* (or forbidden). Secondly, the memory is used to help intensify the search of *good* areas of the search space.

The seminal work on the tabu search method appears in [Glo86]. The later work by Glover, *et al.* gives details of method and of the various enhancements and improvements suggested during the last decade of research [GL93]. The basic technique works by replacing the standard neighbourhood function seen previously with $N(H, \mathbf{x})$. Here, H , represents a selective memory (or history)

of the previously encountered search states. This memory has both short-term and long-term objectives. An outline of the tabu search method is given in figure 4.4

4.4.1 Short-Term Memory – Recency

The objective for the use of this kind of memory is to encourage exploration of previously unvisited parts of the solution space. This is achieved by prohibiting the reversal of previous moves. However, such an absolute constraint would restrict the search too greatly. Thus in tabu search only *recent* moves are prohibited (*tabu*). The length of time or number of iterations for which a moves reversal is *tabu* is known as the *tabu-tenure*. This can remain fixed or can vary dynamically during the search. Empirical evidence presented in [Ree96] suggests that a value of around 7 is often sufficient to prevent cycling and thus makes a good *tabu-tenure*. Other common value include the use of \sqrt{n} where n is some measure of the problem size. Dynamic rules which allow the tabu tenure to vary between a minimum and maximum value during the execution of the search have also been shown to be effective [GTdW93].

Often, rather than maintaining information about complete moves, the focus is placed on components of a move known as *move-attributes*. During progression of the search these attributes can become *tabu-active*. It is then another implementation decision as to how many *tabu-active* attributes a move must have to become *tabu*. Obviously the selection of the attributes is vital to the success of the search. Some illustrative examples of good candidates for move attributes for problems involving binary decision variables are shown below [GL93].

1. Change of a decision variable from 0 to 1
2. Change of a decision variable from 1 to 0
3. Combination of 1 and 2
4. Change of $COST(x_{current})$ to $COST(x_{trial})$
5. Change of a function $g(x_{current})$ to $g(x_{trial})$
 - Where g is some problem specific function.
6. Value of $g(x_{current}) \Leftrightarrow g(x_{trail})$
7. Combination of 5 and 6 for more than one function g

There may, however, be cases when the *tabu-status* of a move overly constrains the search. This can be overcome by the use of an *aspiration* criteria which allows a move's or move attribute's tabu status to be overruled. The simplest aspiration criteria is that a move which improves on the best solution found so far should never be *tabu*. This can be justified simply because the objective of using recency-based memory is to avoid repeated search of the same area of the solution space. However, if a solution represents the best solution found so far during the search then it is obvious that this area of the solution space has not yet been explored.

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

```
procedure Tabu_Search is

  Current_Solution, Best_Solution, New_Solution : Solutions;
  C_Neighbour : Set_Of_Solutions;
  Best_Cost    : Float;
  History_List : List_Of_History_Records;

begin

  INITIALISE (Current_Solution);           10
    -- Start with random solution

  Best_Cost    := CALC_COST (Current_Solution);
  Best_Solution := Current_Solution;
    -- Store information on best solution so far

  loop

    C_Neighbour := CALC_CANDID (Neighbour (History_List, Current_Solution));
    -- Candidate neighbourhood is a subset of the           20
    -- neighbourhood of the current solution

    New_Solution := MIN_SOLUTION (History_List, CALC_COST (C_Neighbour));
    -- The new solution is the solution from the
    -- candidate neighbourhood which has the minimal
    -- cost, taking account of the history list

    History_List := UPDATE (History_List, New_Solution, Current_Solution);
    -- Information about about the attributes of the
    -- change from the current solution to the           30
    -- new solution are added to the history list

    if CALC_COST (New_Solution) < Best_Cost then
      Best_Cost := CALC_COST (New_Solution)
      Best_Solution := New_Solution;
    end if;

    Current_Solution := New_Solution;
    -- Move on to search the candidate neighbourhood of
    -- the new solution                                     40

  exit when STOP_CRITERION;

end loop;

end Tabu_Search;
```

Figure 4.4: Tabu Search Algorithm

4.4.2 Long-Term Memory – Frequency

While recency-based memory is used to further the short-term objectives of the search, frequency-based memory is used as part of the longer-term search strategy. Frequency-based memory measures the frequency with which some move (or move attribute) occurs. This is normally expressed as a fraction with the number of occurrences of the move (or move attribute) as the numerator and one of the following denominators [GL93].

1. The total number of occurrences of all events represented by the numerator.
2. The sum of all numerators.
3. The maximum numerator value.
4. The average numerator value.

This ratio is then used to modify the objective (cost) function to give $c(x_{trial}, F)$. This modification can be used to penalise frequently seen moves (or move attributes) which in turn encourages diversification in the search. The frequency measures can also be used to record the attributes present in *good* (or *elite*) solutions, this information can be used to encourage the selection of solutions which contain these attributes. This leads to the intensification of the search around areas of good solutions.

4.4.3 Neighbourhood Exploration

The tabu search algorithm requires that the best solution from a given neighbourhood is always found (taking into account any tabu-restrictions or aspiration criteria). In practice, however, it may not be computationally feasible to explore an entire neighbourhood in the search for the best solution. In this case a candidate list procedure is used to find a candidate subset of moves from the large neighbourhood. If the particular problem allows, it is possible to strategically sample the neighbourhood in areas where good solutions are likely. In general, however, a simple random sampling of the neighbourhood to obtain a candidate list can be performed. If no suitable solutions are present in the candidate list then another one can be generated.

The use of candidate lists has a three-fold impact on the algorithm. Firstly, it reduces the computational expense involved in testing all of the neighbourhood solutions (which in many cases may prove to be computationally infeasible). Secondly, it automatically has a diversifying effect on the search. This is because the search is moved into areas of non-optimality because only a limited sampling of the neighbourhood is being used to select the search direction. Finally, it introduces a randomised effect into the search. This may not be a problem as there are many other ways in which randomisation is introduced into the algorithm anyway, such as the random selection of a tabu-tenure between two given values during the search.

4.4.4 Example

Figure 4.5 shows the application of tabu-search to the problem outlined in section 4.2.1, the same definition for the neighbourhood structure is used. To

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

increase the efficiency of the tabu-search a candidate list procedure is used to select the subset of the neighbourhood used at each iteration of the search. For this example the candidate list procedure simply randomly selects four of the solutions from the neighbourhood.

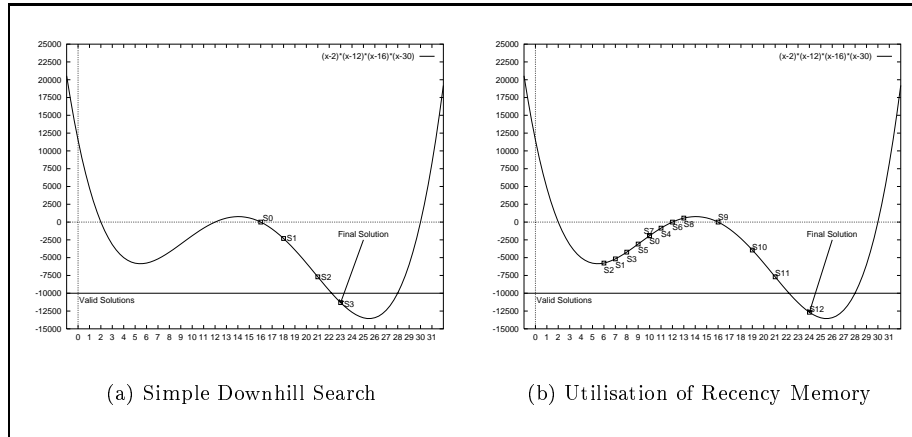


Figure 4.5: Tabu-Search Example

The progress of a simple tabu-search is shown in figure 4.5a and described in the following table. From the initial solution S_0 the search proceeds downhill to the final solution S_3 which solves the constraint. In this case the tabu-search behaves exactly the same as a simple random descent neighbourhood search. This is due to the direct downhill path from the initial solution to the final solution.

Iteration	Candidate Solutions	$f(x)$	Tabu Status	New Solution
Initial Solution = 16, Move to 16 tabu until iteration 4.				
1 (16)	18	-2304		←
	15	585		
	17	-975		
	13	561		
Move to 18 tabu until iteration 5.				
2 (18)	15	585		
	17	-975	Tabu	
	16	0	Tabu	
	21	-7695		←
Move to 21 tabu until iteration 6.				
3 (21)	20	-5760		←
	23	-11319		
	19	-3927		
	18	-2304	Tabu	
Accept solution 23 and stop search				

The ability of tabu-search to overcome the weaknesses of simple neighbourhood search is shown in figure 4.5b. In this case the direct downhill path leads

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

only to a local optimum that does not solve the constraint. As the search progresses from the initial solution S_0 into the local optimum, the recency memory makes move reversals tabu. This prevents the search continually sliding back into the local optimum and allows it to progress to a better solution S_{13} which does solve the constraint.

Iteration	Candidate Solutions	$f(x)$	Tabu Status	New Solution
Initial Solution = 10, Move to 10 tabu until iteration 7.				
1 (10)	7	-5175		←
	9	-3087		
	12	0		
	11	-855		
Move to 7 tabu until iteration 8.				
2 (7)	8	-4224	Tabu	←
	10	-1920		
	6	-5760		
	9	-3087		
Move to 6 tabu until iteration 9.				
3 (6)	3	-3159	Tabu	←
	7	-5175		
	8	-4224		
	9	-3087		
Move to 8 tabu until iteration 10.				
4 (8)	10	-1920	Tabu	←
	6	-5760	Tabu	
	11	-855		
	7	-5175	Tabu	
Move to 11 tabu until iteration 11.				
5 (11)	13	561	Tabu	←
	9	-3087		
	10	-1920		
	12	0		
Move to 9 tabu until iteration 12.				
6 (9)	12	0	Tabu	←
	6	-5760		
	10	-1920		
	7	-5175		
Move to 12 tabu until iteration 13.				
7 (12)	11	-855	Tabu	←
	10	-1920		
	14	768		
	15	585		
Move to 10 tabu until iteration 14.				

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

Iteration	Candidate Solutions	$f(x)$	Tabu Status	New Solution
8 (10)	8	-4224	Tabu	
	13	561		←
	11	-855	Tabu	
	9	-3087	Tabu	
	Move to 13 tabu until iteration 15.			
9 (13)	10	-1920	Tabu	
	11	-855	Tabu	
	16	0		←
	12	0	Tabu	
	Move to 16 tabu until iteration 16.			
10 (16)	17	-975		
	14	768		
	13	561	Tabu	
	19	-3927		←
	Move to 19 tabu until iteration 17.			
11 (19)	16	0	Tabu	
	20	-5760		
	18	-2304		
	21	-7695		←
	Move to 21 tabu until iteration 18.			
12 (21)	22	-9600		
	19	-3927	Tabu	
	20	-5760		
	23	-11319		←
	Accept solution 23 and stop search			

4.5 Genetic Algorithms

The seminal work on genetic algorithms was performed by Holland and his team at the University of Michigan in the 1960s and 1970s, [Hol75]. Genetic algorithms attempt to make intelligent use of the information which is gathered during random sampling of the solution space. Earlier attempts to do the same, [RF66], resulted in methods that were inflexible and very problem specific. Genetic algorithms however represent a general framework which can be applied to many problems.

Genetic algorithms are so called because of their attempt to model the natural genetic evolutionary process. The basis for this is the representation of complex structures using a vector of components (known as a *chromosome*). Genetic algorithms then model selective breeding to obtain offspring which have characteristics inherited from each parent, just like evolution in nature. Normally *chromosomes* are simply binary strings. Goldberg [Gol89] suggests that this has significant advantages even if it is not a natural representation for the problem domain. More recent empirical evidence [Ant89] cast some doubt on this. However, much of the theoretical developments in the application of genetic algorithms are based on the use of binary strings for *chromosomes*.

Genetic algorithms work by maintaining a *population* of *chromosomes* each

of whose *fitness* value has been calculated⁶. Successive populations (known as *generations*) are evolved using genetic operations. The aim is that through the use of the genetic operations the *population* will converge towards a solution. An outline of the genetic algorithm search process is shown in figure 4.6

In Holland's original genetic algorithm the parents were selected based on their *fitness* value (i.e. with a probability proportional to their *fitness* value). The parents are then combined (*mated*) using the *crossover* operator. This operator selects a random crossover point X. The new *offspring* then consist of pre-X section from one parent combined with the post-X section of the other parent. One of the original population is then selected at random and replaced by one of the offspring. The mutation operator introduces new genetic information into the gene-pool by randomly mutating chromosomes with some small probability. There are many alternative variations to this original scheme. The remainder of this section looks at these alternatives and advancements to the simple genetic algorithm.

4.5.1 Encodings

Obviously the encoding strategy is central to the successful application of genetic algorithms. Most genetic algorithm implementation described in the literature use fixed-length, fixed-order bit strings to encode solutions. More recently, empirical evidence has begun to show other encodings can be used successfully. However, the theoretical analysis of such encodings is not yet fully developed.

The use of binary encodings stems from the historical development of genetic algorithms. Holland's work focused on the use of binary encodings (especially in the development of *schema* theory); it also gave a theoretical justification for using a binary encoding. It was shown that a binary encoding, when compared with a larger alphabet encoding with the same information carrying capacity, contains more schemas. It is these schemas which give genetic algorithms their property of implicitly parallel search of the solution space.

One problem with a binary encoding is that solutions which are close in the original search space can be very far apart in the binary encoded space (for example, consider a 6-bit chromosome representing 32 - 1 0 0 0 0 0, a small move to 31 in the solution space means a very large move in the binary space to 0 1 1 1 1 1). This led Caruana, *et al.* [CS88] to suggest the use of Gray code mapping. This is particularly relevant in cases where sequence representation is necessary but the implementation is complicated by there being no simple algorithm for decoding Gray code.

For many problems for which a binary or Gray code encoding is unnatural many-character and real-valued encodings can be used. This more natural encoding scheme can give significant performance improvements for the particular problems [JM91]. New versions of the genetic operators can then be devised to fit the encoding used more effectively, as discussed in [Dav91]. Other problems can most naturally be expressed as a permutation of a set of numbers (for example, the travelling salesman problem). In this case again new genetic operators have to be devised as the standard *crossover* operator will result in offspring which are not permutations. A simple inversion operator can be applied to

⁶As each *chromosome* represents an encoding of a solution to the underlying problem a *fitness* value can be calculated as related to the *cost/objective* function for the given solution.

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

```
procedure Genetic_Algorithm is

    Current_Population, Parents, Offspring :
        array (1..N) of Chromosomes;

begin

    INITIALISE (Current_Population);
        -- Assign initial values to the population
                                                                    10

    CALC_FITNESS (Current_Population);
        -- Calculate the fitness for each member of the
        -- population, related to the objective function
        -- for the solution represented by the chromosomes

    loop

        SELECT_PROPECTIVE_PARENTS (Current_Population, Parents);
            -- Select which members of the population are to be used
            -- as parents
                                                                    20

        RECOMBINE (Offspring, Parents);
            -- Combine parents to produce offspring

        MUTATE (Offspring);
            -- Stochastically mutate the new offspring to introduce
            -- diversity

        CALC_FITNESS (Offspring);
                                                                    30

        Current_Population :=
            SELECT_NEW_POPULATION (Current_Population, Offspring);
            -- Select which of the original population and the
            -- offspring will survive to the new population

        exit when STOP_CRITERION;
    end loop;

end Genetic_Algorithm;
```

Figure 4.6: Genetic Algorithm

single strings⁷. However, the strength of genetic algorithms comes from their ability to combine aspects of different chromosomes. Suitable operators have been the subject of work in [GL85], [Gol89] and [Ree92] (see section 4.5.3).

At present there is no theory which enables a rigorous approach to the selection of the best encoding method for a particular problems. Davis [Dav91] suggests that the best approach is to use the encoding which is most natural for the particular problem domain and then to devise genetic operators which are suitable for the selected encoding.

4.5.2 Selection Methods

The selection method involves devising a way in which to choose both the parents which will create the offspring and which chromosomes will survive into the next generation. In Holland's original work the parents were selected probabilistically according to fitness and a complete new population of offspring was produced to form the new generation. This is known as *roulette wheel* sampling. This can be implemented by assigning a cumulative range to each parent between 0 and 1 and then generating N random numbers between 0 and 1 to select which individuals will become parents (where N is the number of individuals in the population). This method can cause problems because of the relatively small population sizes involved in genetic algorithms. Because of the random nature of the selection, sampling errors can lead to serious differences between the actual and expected number of times which an individual will become a parent.

Sigma scaling, developed by Forrest [Mit96], is one method to overcome these sampling errors. Using this scheme for selection an individual's expected offspring count is a function of its fitness, the population mean fitness and the population fitness standard deviation. At the start of the search the standard deviation will be large and thus fit individuals will not lie many standard deviations from the population mean and hence will not overrun the population. However, as the search progresses and the standard deviation falls as the population converges, the search will be focused more towards the fit individuals.

Elitism is a method which allows highly fit individuals to survive from one generation to the next without hindrance from the parental selection process or the genetic operators. The ideas were introduced in [Jon75] and have been shown on many occasions to significantly increase the performance of the genetic algorithm search.

Rank selection, proposed by Barker [Bar85], abstracts away from absolute fitness values and simply ranks individuals based on their fitness value. These ranks are then used to determine the expected offspring for each individual. This has the effect of slowing down convergence as highly fit individuals do not stand out significantly in the selection process as they are only one rank better than their nearest rival.

Tournament selection is a computationally more efficient method which gives results similar to that of rank selection. The basic idea is to randomly select two individuals from the population. A random number is then generated and, if greater than a user-defined parameter, the fitter of the two individuals becomes

⁷The inversion operator simply reverses the order of the values between two points in the chromosome (string).

the parent otherwise the less fit individual becomes the parent. Both are then returned to the original population and may be selected again.

Most implementations of genetic algorithms have been generation based, meaning that each new population consists entirely of offspring formed by parents from the previous generation. Strategies such as elitism allow successive generations to overlap to some degree. The idea of a generation gap was introduced by De Jong, [Jon75]. This represents the proportion of new individuals in each new generation. In steady-state selection (also known as incremental selection) only a very few individuals are replaced in each generation. This method has been used with some success. One particular advantage is that it makes it easier to avoid duplicates within a generation. Duplication can cause problems as it wastes effort evaluating the duplicate and also distorts the selection process in favour of the duplicated individual [Dav91].

4.5.3 Genetic Operators

The main genetic operator is the crossover operator. The use of the simple single-point crossover operator can destroy information contained particularly in long chromosomes. This fact led Booker [Boo87] to suggest the introduction of multi-point crossover, which has been shown to increase the performance of a genetic algorithm.

The secondary genetic operator is the mutation operator. This is applied at a low probability to introduce diversity into the search.

Davis [Dav91] discusses the possibilities when dealing with non-binary encodings. The concepts of an *averaging crossover* operator and a *creep mutation* operator are introduced. For problems encoded as a permutation Goldberg and Lingle [GL85] defined a partially mapped crossover operator which uses two crossover points. Between these points an interchange mapping is defined. In this way the operator preserves the permutation property. Reeves [Ree92] defines an exchange mutation operator which simply swaps two random elements of the individual. However, this was found to produce less efficient genetic algorithms when compared with a mutation operator which simply shifted an element a number of places left or right.

4.5.4 Parameters

The parameters of a genetic algorithm include the population size, crossover rate, mutation rate, generation gap and the selection strategy parameters. Work by Grefenstette [Gre86] attempted to use a genetic algorithm to optimise the parameters to another genetic algorithm with some success. The most promising approach to determining parameter values at present is to have the values adapt as the search progresses. Research in this area is still very much in its infancy.

4.5.5 Example

Figure 4.7 shows the application of a genetic algorithm to the example problem used throughout this chapter. The genetic algorithm used is extremely simple but, as can be seen below, is still effective. The method of encoding simply uses a 5-bit binary string. The population size is 5 and the initial population contains randomly selected solutions. Crossover is applied with a probability of 1 at a

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

randomly select bit position. Mutation is applied with a probability of 0.02 at each bit position. One of the parents is selected using tournament selection (between two randomly selected candidates, the fittest of which becomes the parent) and the other parent is selected randomly.

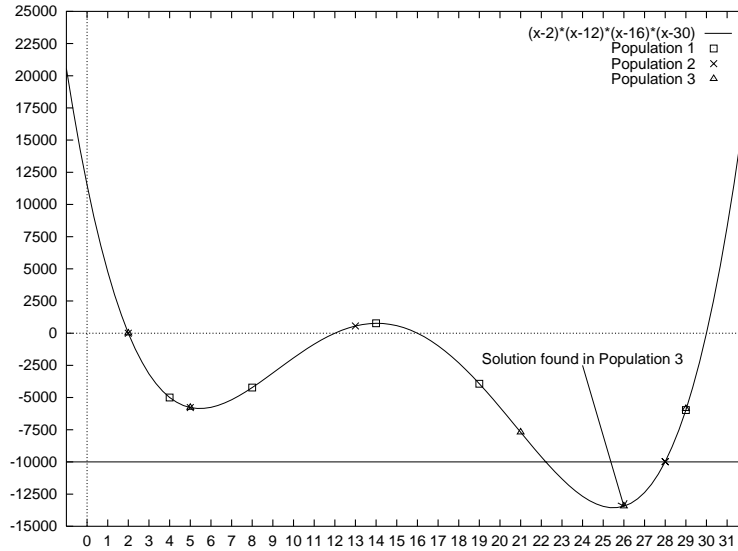


Figure 4.7: Genetic Algorithm Search Example

The following table shows the progress of the search from an initial population of random solutions to a population containing a solution to the constraint (shown in bold). It can be seen that the overall average of each successive population is increasing as the genetic algorithm converges towards a solution.

Initial Population		Parents		Offspring				
T_1	T_2	P_1	P_2	Cross	Mutation	x	String	$f(x)$
						4	01101	-4992
						8	01000	-4224
						14	01110	768
						19	10011	-3927
						29	11101	-6967
						Average Fitness = \approx 3668.4		

CHAPTER 4. HEURISTIC OPTIMISATION TECHNIQUES

Second Population								
Tournament		Parents		Cross	Mutation	Offspring		
T_1	T_2	P_1	P_2			x	String	$f(x)$
8	14	8	29	1	NNNNN	13	01101	561
19	4	4	14	2	NNYNN	2	00010	0
29	4	29	19	4	NNNNN	29	11100	-9984
4	19	4	29	4	NNNNN	5	00101	-5775
19	29	29	4	2	NNNNN	28	11100	-9884

Average Fitness = $\Leftrightarrow 5036.4$

Third Population								
Tournament		Parents		Cross	Mutation	Offspring		
T_1	T_2	P_1	P_2			x	String	$f(x)$
5	2	5	2	2	NNNNN	2	00010	0
28	28	28	13	3	NNNNN	29	11101	-5967
28	5	28	5	1	NNNNN	21	10101	-7695
13	2	2	5	3	NNYNN	5	00101	-5775
13	28	28	2	2	NNNNN	26	11010	-13440

Average Fitness = $\Leftrightarrow 6575.4$

Part III

Future Work and Preliminary Results

Chapter 5

Conclusions and Proposal

This report has reviewed both the history and state-of-the-art in automated software testing and optimisation techniques. The following presents the conclusions and ideas for further work in these areas.

Automated Software Testing

The current approaches in the literature are too inflexible and have limited capacity. They are often limited to particular data-types (often only integers) or control-flow structures. These restrictions make them unsuitable for general application. Each approach to automating test-data generation focuses on generating test-data for single criteria. No attempt has been made to develop a general framework within which many testing criteria can be met, especially in the area of unifying the testing of both functional and non-functional properties.

Optimisation in Automating Software Testing

Optimisation techniques have shown themselves to be a flexible and powerful approach to solving *difficult* problems. Despite the suggestion by Miller in 1976 that global optimisation techniques represented a way forward for automating test-data generation, work has only recently begun to appear in the literature. The efficiency of optimisation techniques for test-data generation may be improved significantly with modifications and enhancements to the algorithms. For example, modifications to the neighbourhood structure of simulated annealing or careful analysis of the effectiveness of move attributes for tabu search. While such heuristic optimisation techniques will never be able to guarantee their results, it may be possible to devise metrics which, given the software to test, give an indication of which technique will be best, or even suggest settings for the technique's parameters. Also there is significant scope to improve the current state-of-the-art despite the lack of guaranteed results.

Given the generality of optimisation techniques, it is hoped that they can be used to give a general framework which can be used to generate test-case data for a wide class of testing criteria. Examples of such testing criteria which may fit into the general framework include:

- Functional
 - Coverage Criteria
 - Mutation Analysis
 - Specification Failure
 - Boundary Analysis
- Non-Functional
 - Worst-Case Execution Time
 - Stack Usage
 - General Resource Usage
 - Environmental Testing - Precision and Capacity
- Management
 - Optimal Test-Case Subsets For Multiple Criteria
 - Regression Testing Subsets

Proposal For Further Work

This section details the areas of further work which will need to be addressed in order to demonstrate feasibility of an optimisation-based general approach to test-data generation.

Optimisation Techniques

An investigation into methods of applying various optimisation techniques to the software testing domain is needed. A preliminary study of how to approach this problem has already been completed and is presented in the following chapter. However, a detailed comparison of the various optimisation techniques to discover their strengths and weaknesses is required. This will involve the investigation of specific enhancements and tailoring of the methods to improve their performance for particular testing criteria.

It is envisaged that this work will take between three and six months. This will produce guidelines as to which methods are suitable for which testing problems. Also a number of testing specific enhancements and optimisations will be produced. The aim for this task is to culminate in the production of a paper suitable for an optimisation oriented journal.

Testing as Optimisation

For the set of testing problems under consideration (not all of which have been decided) it will be necessary to consider approaches to representing each as an optimisation problem. For a small number of testing problems this has already been carried out as presented in the following chapter. However a great deal more work is required. For example, certain cost function may give better search guidance or more efficient stop-conditions. Each testing problem will need to be considered in turn and the alternatives explored. This process will take

CHAPTER 5. CONCLUSIONS AND PROPOSAL

upwards of three months, obviously depending on how many testing problems are considered.

This work will produce a set of methods and cost functions which will allow optimisation techniques to be used to generate test-data for a various testing problems. It is hoped that this work will form the basis for a conference paper in automated software testing.

Metrics

As optimisation techniques cannot guarantee a result methods of determining the quality of test-data obtained will be needed. For many problems the quality of the test-data will be easily determined. For example, when attempting to find test-data which shows non-conformance to a formal specification, it is trivial to show that the test-data meets the pre-condition and does not meet the post-condition and hence is good quality test-data. However, for other problems it is not so easy to determine the quality of the test-data. For example, when searching for worst-case execution test-data it is not possible to give an absolute measure of the quality of the test-data as if the real worst-case execution time was known then there would be no need for the test. It may be possible to provide a mapping of existing software metrics onto the expected quality of the resulting test-data. The assumption is that more complex routines (as measured by a particular metric) are likely to be more complex to test and hence the resulting test-data may be of a lower quality.

This work will require large scale case-studies and the investigation into currently available software metrics. This combined is likely to take upwards of nine months.

Case-Studies

In order to demonstrate feasibility and to build credibility of the optimisation based approach to software testing empirical evidence will be required. Too many approaches to automated software testing presented in the literature give little or no evidence of the wider capabilities or limitations of the approach. This will require applying the techniques to large scale case-studies and comparing the results with the current state-of-the-art and state-of-practice in software testing. More consideration of how to obtain the required source code is required.

Combined with the development of metrics, this work is likely to take upwards of nine months. The aim is to produce sufficient evidence to demonstrate the efficacy of the approach and further research. The aim is for this work to culminate in a summary paper presenting the approach and empirical evidence.

Prototype Tool-Set Framework

In order to facilitate the development of the above research it will be necessary to produce a tool-set which allows the various optimisation techniques, cost-functions and metrics to be explored. This development will proceed concurrently with the above tasks at a rate sufficient so as to allow results to be obtained at each stage. Some preliminary development has already been carried out and is presented in the following chapter.

Assessment

This work will be judged by several criteria. In terms of originality, there is the development of a general framework for generated test-data for both functional (*black-box* and *white-box*) and non-function testing criteria. No previous attempt to unify automated testing in such a manner has been reported in the literature. While work on optimisation as an approach to generating test-data has just begun to appear, this field is still young, therefore there is scope to follow and influence the developments and advances. However, the work will mainly be judge by the effectiveness of the end-product – i.e. can high-quality test-data be efficiently generated for a number of testing criteria? If successful, it is hoped such an approach to automating test-data generation will allow improved software testing with reduced effort.

Chapter 6

Preliminary Evaluation and Results

This chapter presents the preliminary work already undertaken. This work has been two-fold. Firstly, a preliminary examination of how to apply the optimisation techniques (presented in chapter 4) to software testing problems. Secondly, the basis of a prototype tool set has been developed to explore the feasibility of the optimisation approach to software testing. The following section outlines the approaches for describing software testing problems as optimisation problems. This is followed by a review of the current capabilities of the prototype tool-set and some of the preliminary results obtained using it.

6.1 Automated Software Testing as Optimisation

For many (if not all) software testing criteria there is the concept of a *good* test-case – that is a test-case which is better than some other test-cases. Generally it is the software tester’s responsibility to find the best test-case (or test-set) within particular constraints (usually time or financial constraints). Searching for near-optimal solutions in a difficult search space is what optimisation techniques do best. Despite this important similarity, there are still a number of questions to be answered before optimisation techniques can be used in automating test-case data generation. These questions fall into two classes – those which are specific to the optimisation technique but reasonably independent of the testing criteria and those which are independent of the optimisation technique but specific to the testing criteria. The sorts of questions in the first class include how to define a neighbourhood, how to encode solutions, what solution attributes are important, etc. These issues will be discussed in the following sections for each of the optimisation techniques: simulated annealing, tabu-search and genetic algorithms. The second class of questions is mainly concerned with measuring how *good* a particular solution is for the given testing criteria. Approaches to this are discussed for a number of software testing criteria.

6.1.1 Simulated Annealing

The first consideration must be to determine exactly what is a solution to a software testing problem. Generally, the solution will be some data values which represent a test-case (or test-set). This leads to the natural representation of a solution as a number of data items which are derived from the underlying data types of the programming language used for the software under test. As simulated annealing is a neighbourhood search strategy, this concept must be defined. The neighbourhood should represent the set of solutions which is in some respect *close* to a given solution. Given the representation of solutions, the neighbourhood can be defined as shown in table 6.1 for the fundamental Ada types¹.

Basic Type	Neighbourhood
INTEGER	\pm Some proportion of allowed range
FLOAT	\pm Some proportion of allowed range
BOOLEAN	TRUE or FALSE
ENUMERATION	Any value from the enumeration

Table 6.1: Neighbourhood Structure for Ada Types

As the simulated annealing search progresses, the size of this neighbourhood could be reduced to help prevent jumping out of a good area of the search space during the later stages of the search. The generation mechanism for simulated annealing can then be simply defined as the selection of a random solution from the neighbourhood of the current solution.

Many of the enhancements discussed in 4.3.4 may improve the search efficiency significantly. Also, the cooling schedule can have a dramatic impact on performance. A detailed examination of these issues will be performed as part of the work on evaluating the performance of the optimisation techniques.

6.1.2 Tabu Search

The first consideration, again, is the representation of a solution. As for simulated annealing, the most natural representation is a number of data items derived from the underlying data types which represent a test-case (or test-set). Tabu search, like simulated annealing, is a neighbourhood based search strategy. Thus the neighbourhood can be defined as for simulated annealing (shown in table 6.1). Due to the potential size of the neighbourhood and the cost of an exhaustive neighbourhood search, a candidate list procedure will be required. This can simply sample the neighbourhood to obtain a given number of candidate solutions.

To implement the tabu search concept of memory it is necessary to define the move attributes which are suitable. Suitable move attributes must aim to prevent the reversal of moves and to push the search into new, unexplored areas of the search space. Some sample candidate move attributes are described below.

¹During the development of the prototype, Ada has been used as the language for the software under test.

1. Increase in value of a numeric variable
2. Decrease in value of a numeric variable
3. Combination of 1 and 2 for different variables
4. Change of a binary variable from TRUE to FALSE
5. Change of a binary variable from FALSE to TRUE
6. Combination of 4 and 5
7. Change in value of an enumeration variable
8. Change in numeric variables X and Y from $X > Y$ to $X < Y$
9. Change in numeric variables X and Y from $X < Y$ to $X > Y$
10. Change in numeric variables X and Y from $X = Y$ to $X \neq Y$
11. Change in numeric variables X and Y from $X \neq Y$ to $X = Y$

Only experimentation will determine the best level at which to set parameters such as the tabu tenure and number of tabu attributes necessary before a move becomes tabu. The simple aspiration criteria of always accepting moves to solutions which are better than any other visited solutions may also prove useful in improving the search.

6.1.3 Genetic Algorithms

A number of different solution encodings present themselves as suitable for genetic algorithms. These encoding methods are discussed below.

Simple Data Values – As for simulated annealing and tabu search. This requires changes to the crossover and mutation operators. Suitable changes have been suggested in [Dav91] in which the concepts of an averaging crossover and creep mutation operator are defined. Averaging crossover, as implied by the name, selects crossover points and then averages the values between these points. The creep mutation operator changes the value of a variable by a random amount from a given range. This range can be reduced as the search progresses to prevent big jumps away from good solutions.

Binary Encoding – Allows data type generality as all data types are represented in the same format. This encoding method is also the most common for genetic algorithms and hence has the greatest research to draw upon. However, problems exist in that neighbouring solutions in the solution space may not be neighbouring in the encoding space due to the encoding. This can result in a slow, inefficient search as the degree of guidance is reduced. Also genetic operators (mutation and crossover) can result in bit-patterns which are invalid for the particular data-type they represent. This problem requires either that such bit-patterns are detected and discarded or that the genetic operators be modified to become *smarter* and more aware of the problem. The crossover operator must also be modified to only crossover at points which represent the boundary between two variables in the encoded solution.

Gray Code – This improves on the neighbouring solution problem from binary encodings. However the illegal bit-pattern problem still remains.

Again, as with tabu search, only experimentation will determine the best values for population size, genetic operator application rates and the most effective selection methods.

6.1.4 Objective/Cost Functions

In order to guide the optimisation search it is necessary to be able to calculate a measure of *goodness* for each solution examined. Obviously, this value is dependent entirely upon the testing problem being addressed. The following examines ideas for cost functions for a number of test-data generation problems, some of which have been implemented in the prototype tool-set.

Constraint Solving – The objective function devised here requires the constraint system to be converted to Disjunctive Normal Form (DNF). This allows each disjunct to be analysed in turn, as a solution to any one disjunct represents a solution to the entire constraint system. The cost of all the terms within the disjunct are summed to give the overall cost. The value for each term is calculated as shown in table 6.2. If the satisfaction criterion is not met then the value of the term is given by the function, otherwise the term adds nothing to the overall cost.

Term	Function	Satisfaction Criteria
TRUE	None	Always Satisfied
FALSE	None	Always Unsatisfied
Boolean Variable	None	Depends on variable value
$a = b$	$abs(a \leftrightarrow b)$	$= 0$
$a \neq b$	$abs(a \leftrightarrow b)$	$\neq 0$
$a < b$	$a \leftrightarrow b$	< 0
$a \leq b$	$a \leftrightarrow b$	≤ 0
$a > b$	$b \leftrightarrow a$	< 0
$a \geq b$	$b \leftrightarrow a$	≤ 0

Table 6.2: Cost Function Values for Terms

This cost function defines the additional stopping criterion of when a cost of zero is found. This increases the efficiency of the search.

Timing Analysis – The objective function here simply becomes a measure of the execution time of the unit.

Resource Usage – The objective function is a measure of the amount of resource used.

Path Based Testing – Path based testing is concerned with the execution of a particular path through, or statement of, a software unit. One approach for devising an objective function for this problem is to produce a straight-line version of the particular path. This straight-line version reproduces

the same code except that conditional statements are replaced with functions. These functions are the same as the objective function for constraint solving. The values of each of these functions are then summed. This resulting value will only be zero when all of the conditional statements in the path are satisfied, hence test-case data have been found which execute the desired path. An alternative approach would be to base the objective function on the work by Jones [JSE96] (see section 2.2.4). Using this approach, the objective function value is related to the branch predicate when a sibling of the desired control-flow node is executed, otherwise it is set to some default high value.

Mutation Testing – The objective function here is simply a combination of a path based function and a constraint function. Killing mutants requires that two conditions be satisfied. Firstly the mutated statement must be executed (path-based testing). Secondly, the necessary mutation condition must be satisfied (constraint solving).

Specification Testing – A software unit’s specification can be represented as a pre-condition constraint on input values and a post-condition constraint on input/output values. Using this specification, test-cases can be derived which either show conformance to the specification or non-conformance. The objective function is basically the same as that for constraint solving. The pre- and post-condition are converted to DNF and all pairs of pre- and post-condition conjunctions are formed and considered in turn. The input values and output values (i.e. values after the test-unit has been executed) for a given solution can be stored. The value of the objective function is then calculated using these values and the pre/post-condition disjunct pair. Obviously, when deriving test-cases to show that the specification is not met, the post-condition is negated before the conversion to DNF. Thus the search is then for a test-case which satisfies the pre-condition, but falsifies the post-condition.

Bounds Testing – This is basically the same as constraint solving. The values used may be intermediate values at a particular point of execution of the software under test. For this reason the calculation of the objective function may need to be embedded within the software under test. The constraint would be formed according to the bounds which require testing. For example to see if X went outside the bound 0..100 the constraint would be $X < 0 \vee X > 100$.

Exception Testing – The SPARK examiner can generate exception-free verification conditions [Pra95]. One approach to exception testing could be to try to find a test-case which falsifies these conditions and hence raises an exception. The objective function in this case would be identical to the constraint solving objective function.

Test Set Management – Optimisation could help in the trade-offs involved in testing management such as test confidence, effort, budget and time constraints.

6.2 Feasibility Prototype Tool-set

The prototype tool-set developed so far consists of a number of free-standing tools. The communication between the tools is via ASCII files which have defined syntax and semantics. This allows the tools to be modified and expanded with minimal inter-tool dependencies. The tool-set is currently capable of performing random or simulated annealing search for test-data. The testing criteria addressed at present are:

- Test-Case with Worst-Case Execution Time (WCET)
- Test-Case with Best-Case Execution Time (BCET)
- Test-Case for Specification Satisfaction
- Test-Case for Specification Falsification
- General Constraint Solving

Figure 6.1 shows the tools which make up the tool set. The purpose of each of these tools is briefly discussed below. This is followed by some preliminary results obtained using the prototype tools for each of the above testing problems.

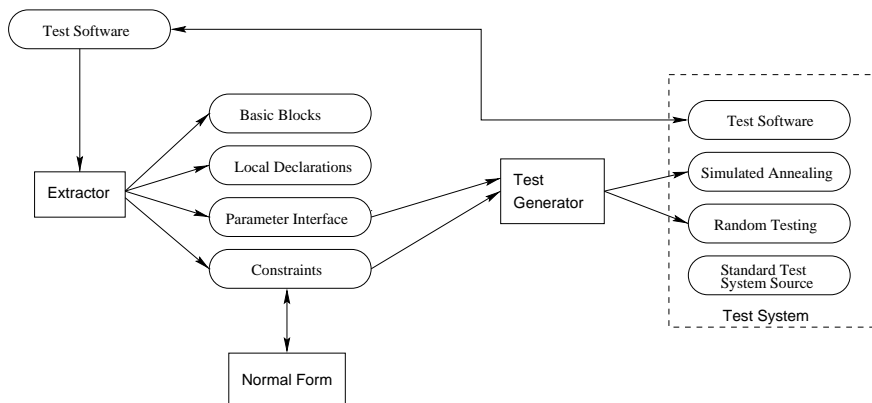


Figure 6.1: Prototype Tool Set

Extractor – This tool is responsible for processing the software under test. It extracts all the information necessary for the rest of the testing system and stores it in files. This tool is based on the GNAT [Inc97] compiler front-end as this gives access to a full semantic tree of the test software. This tool will also be responsible for producing instrumented or altered (i.e. straight-line) versions of the software under test when this is needed for other testing problems.

Normal Form – This tool converts a constraint system into DNF.

Test Generator – This tool uses the information from the other tools to produce the test-system. This test system is compiled and linked with the

software under test. When executed, this test system will use the desired search strategy (at present either random or simulated annealing) to find a solution to the desired testing problem.

6.2.1 Testing Problems Addressed by Prototype

Execution Time

To test the feasibility of an optimisation based approach to execution time testing the prototype has been developed to search for both WCET and BCET. The cost function for the search is simply a measure of the execution time of the software being tested. This cost is either maximised (for WCET) or minimised (for BCET). The execution time is measured using a system call to a modified Linux [Rus97] kernel. The modified system call returns information about the actual execution time used by the software. This helps to reduce the effects of varying load. The modifications also increase the resolution of the kernel timing measurements from 10ms to 20 μ s. This resolution, however, is still not good enough. To overcome this problem, the execution of the software is slowed down. This does not affect the test-data found by the system. Once this test-data has been found the actual execution time of the original software can then be measured using this data. Some preliminary evaluation of the system's ability to find test-case data which exercises the worst case path has been carried out. The following programs have been used in these tests.

Conditional Blocks – This software unit takes in two parameters – one integer and one enumeration. It has two conditional blocks, the second of which is dependent upon the decision taken at the first. This demonstrates the system's ability to find valid test-cases in a situation where simple syntax-based analysis would fail.

Simple Loop – This is a simple software unit which has only one integer parameter. This unit iterates around a loop the number of times specified in the parameter. The expected result is the maximal value for the parameter type.

Iterative Integer Square Root – This software unit takes in one integer parameter and returns the integer square root of this value in the output parameter. This routine searches for the integer square root by counting up from 0. Thus the expected result is the maximal value for the parameter type. This demonstrates the system's ability to cope with a discontinuous cost surface.

Binary Integer Square Root – This software unit again calculates the integer square root of its input parameter. However, the algorithm this time performs a binary search for the square root value. Thus the expected result is a value which causes the maximum number of iterations in the binary search. In this case, with the input parameter range from 0 to 10,000, the maximum number of iterations is 14. This demonstrates the system's ability to cope with a still more complex cost surface.

CHAPTER 6. PRELIMINARY EVALUATION AND RESULTS

Insertion Sort – This software unit performs an insertion sort on an array of 5 integer values. The worst case complexity for insertion sort is for reverse sorted data, thus the expected results is an ordered, decreasing array of values. This demonstrates the system’s ability to cope with a large parameter/search space.

The results of the preliminary tests are shown in table 6.3. The system was set to search for the WCET of each of the above programs. The search was repeated 50 times for each program and the results below record the percentage of times a valid test-case was found which exercised the worst-case path.

Program	Size of Parameter Space	Percentage of Parameter Space Searched	Percentage of Valid Test-Cases
Conditional Blocks	1,407	553.1%	100%
Simple Loop	10,000	47.4%	100%
Iterative Square Root	10,000	45.8%	100%
Binary Square Root	10,000	52.4%	100%
Insertion Sort	9.7656e16	7.198e-10%	100%

Table 6.3: Preliminary WCET Results

These preliminary results demonstrate the feasibility of the optimisation approach to generate test-data for WCET. In all cases, the system generated test-case data which exercised the worst-case path. These results show that the method is more efficient with larger, more complex parameter spaces. With small parameter spaces to search, the optimisation based search examined a very large percentage of the parameter space. As the parameter space grows, the number of solutions examined did indeed increase, however the relative percentage fell significantly. The search for test-case data took between 35 seconds and 3 minutes on an Intel Pentium Pro-200MHz.

Specification Testing

The prototype tool-set has been developed to allow it to search for test-case data that either satisfies or falsifies the specification. The cost function for the simulated annealing search is as described in section 6.1.4. Finding test-case data which falsifies a software unit’s specification is normally more interesting, hence the preliminary evaluation of the system has been carried out by searching for a test-case which shows that a software unit does not correctly implement its specification. In each case, the specification appears as part of the source-code as a comment with a similar format to the spark predicate [Pra95]. The following *incorrect* programs have been used in these tests.

Middle – The specification states that given three integer values this routine should return the middle value. If two of the input values are the same then the

CHAPTER 6. PRELIMINARY EVALUATION AND RESULTS

return value should be the other input value. If all three input values are the same then the return value is unspecified. The implementation of this routine, however, always returns the first input value when any of the input values are the same. The expected test-case, therefore, should illustrate this functional error.

Bubble Sort – The specification states that given an array of input values this routine should sort these values into ascending order². The implementation of this routine, however, does not perform enough iterations to sort the array when the smallest input value appears as the last element in the array.

The results of the preliminary tests are shown in table 6.4. The system was set to search for test-data which illustrated a specification failure for each of the above programs. The search was repeated 50 times for each program and the results below record the percentage of times a valid test-case was found which exercised the worst-case path.

Program	Size of Parameter Space	Percentage of Parameter Space Searched	Percentage of Valid Test-Cases
Middle	1000000	0.0445%	100%
Bubble Sort	1e+15	8.711e-11%	100%

Table 6.4: Preliminary Specification-Failure Results

Again these results show the feasibility of the optimisation approach. The prototype tool-set requires a little over 4 minutes to perform the 50 executions of the search for specification failure data for the middle function and a little under 10 minutes for the bubble-sort routine. This demonstrates the feasibility of using such a search as a pre-proof step to prevent proof attempts of code which can be shown to be incorrect using this system.

Constraint-Solving

The prototype tool-set also allows a search for data which satisfies a constraint system. The cost function is again as described in section 6.1.4. A stand-alone version of the constraint-solving search has also been implemented and integrated in to the CADiZ system [TM95]. This is used as the first step in a proof process by generating some test-data which satisfies an existentially quantified constraint. While the constraint-solving system has been used effectively within CADiZ, it does suffer from a number of limitations. Its ability to cope with a large number of dependencies between variables and with finding small solution spaces are perhaps the major problems. Interdependencies between variables (i.e. constraints of the form $X > 7 \wedge Y = 2 \times X \wedge Z = X \times Y$, etc . . .) cause problems because of the minimal probability of the search randomly selecting the correct set of values from the neighbourhood. This problem will be

²In fact, the specification is too weak because it only states that the values in the returned array should be ordered. It does not state that they must be a permutation of the input values.

addressed by future research into the search algorithm. For example, a modified neighbourhood structure, or the use of tabu search with move attributes may help overcome this problem.

6.3 Conclusion

The work carried out to date has seen progress in both research and implementation. Preliminary ideas have been presented on how to apply the optimisation techniques to a software testing domain. Also, methods for representing a number of testing problems as optimisation problems have been given. The efficacy of these ideas and the general approach has been illustrated with the use of a prototype implementation. This has achieved some extremely good results considering the short development time. This combined gives a high-level of confidence that, with further work, it will be possible to meet the objectives and prove, to a large extent, the hypothesis presented at the beginning of this report.

Bibliography

- [Ant89] J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In J. D. Schaffer, editor, *Proceedings of 3rd International Conference on Genetic Algorithms*, pages 86–91. Morgan Kaufmann, 1989.
- [AW95] Alberto Avritzer and Elaine J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *IEEE Transactions of Software Engineering*, 21(9):705–716, September 1995.
- [Bar85] J. E. Barker. Adaptive selection methods for genetic algorithms. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Morgan Kaufmann, 1985.
- [Bat96] Roberto Battiti. *Modern Heuristic Search Methods*, chapter 4 – Reactive Search: Towards Self-tuning Heuristics, pages 61–83. Wiley, 1996.
- [Bei90] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.
- [BEL75] R. Boyer, B. Elspas, and K. Levitt. Select – a formal system for testing and debugging programs by symbolic execution. *Proceedings International Conference of Reliable Software*, pages 234–245, 1975.
- [Boo87] L. B. Booker. *Genetic Algorithms and Simulated Annealing*, chapter Improving search in genetic algorithms. Morgan Kaufmann, 1987.
- [Bur67] W. Burkhardt. Generating programs from syntax. *Computing*, 2(1):83–94, 1967.
- [Bur96] Simon Burton. Automatic test generation. Master’s thesis, Department of Computer Science, University of York, 1996.
- [CBW96] R. Chapman, A. Burns, and A. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11:145–171, 1996.
- [Cla76] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

BIBLIOGRAPHY

- [CpD93] A. Coen-porisini and F. Depaoli. Array representation in symbolic execution. *Computer Languages*, 18(3):197–216, 1993.
- [CS88] R. A. Caruana and J. D. Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In *Proceedings of the Fifth International Conference on Machine Learning*. Morgan Kaufmann, 1988.
- [Dav91] L. D. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [DLS78] R. DeMillo, R Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.
- [DO91] R. Demillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
- [DO93] R. Demillo and A. Offutt. Experimental results form an automatic test case generator. *ACM Transactions on Software Engineering and Methodology*, 2(2):109–127, April 1993.
- [Dow93] Kathryn A. Dowsland. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 2 – Simulated Annealing, pages 20–69. McGraw Hill, 1993.
- [FK96] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.
- [FvBK⁺91] Susumu Fujiwara, Gregor v. Bockmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.
- [GL85] D. E. Goldberg and R. Lingle. Alleles, loci and the travelling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 154–159, 1985.
- [GL93] Fred Glover and Manuel Laguna. *Modern Heuristic Techniques for Combinatorial Problems*, chapter 3 – Tabu Search, pages 70–150. McGraw Hill, 1993.
- [Glo86] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 5:533–549, 1986.
- [Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.
- [Gre86] J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions of Systems, Man and Cybernetics*, 16(1):122–128, 1986.

BIBLIOGRAPHY

- [GTdW93] Fred Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.
- [Hay86] I. Hayes. Specification directed module testing. *IEEE Transactions on Software Engineering*, 12(1):124–133, January 1986.
- [Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
- [Inc87] D. Ince. The automatic generation of test data. *Computer Journal*, 30(1):63–69, 1987.
- [Inc97] Ada Core Technologies Inc. The gnat ada-95 compiler, 1997. <http://www.gnat.com/>.
- [JAMS89] David S. Johnson, Cecilia R. Aragon, L. A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations Research*, 37(6):865–892, November 1989.
- [JM91] C. Z. Janikow and Z. Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*. Morgan Kaufmann, 1991.
- [Jon75] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.
- [JSE96] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.
- [KCDGV83] S. Kirkpatrick, Jr. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [Kin76] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [Kor90] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
- [Kor96] B. Korel. Automated test data generation for programs with procedures. *ACM ISSTA*, pages 209–215, 1996.
- [LM86] M. Lundy and A. Mees. Convergence of an annealing algorithm. *Math. Prog.*, 34:111–124, 1986.
- [Mit96] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.
- [MRTT53] N. Metropolis, A. W. Rosenbluth, A. H. Teller, and E. Teller. Equation of state calculation by fast computing machine. *Journal of Chem. Phys.*, 21:1087–1091, 1953.

BIBLIOGRAPHY

- [MS76] W. Miller and D. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, September 1976.
- [Off91] A. Jefferson Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, 1991.
- [OP96] A. Jefferson Offutt and Jie Pan. The dynamic domain reduction procedure for test data generation. <http://www.isse.gmu.edu/faculty/ofut/rsrch/atdg.html>, 1996.
- [Oul91] M. Ould. Tesintg - a challenge to method and tool developers. *Software Engineering Journal*, 6(2):59–64, March 1991.
- [Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.
- [Pra95] Praxis Critical Systems. *Spark-Ada Documentation 2.0*, 1995.
- [Ree92] C. R. Reeves. A genetic algorithm for flowshop sequencing. *Computers and Operations Research*, 1992.
- [Ree93] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, 1993.
- [Ree96] Colin R. Reeves. *Modern Heuristic Search Methods*, chapter 1 – Modern Heuristic Techniques, pages 1–25. Wiley, 1996.
- [RF66] S. M. Roberts and B. Flores. An engineering approach to the travelling salesman problem. *Man. Sci.*, 13:269–288, 1966.
- [RHC76] C. Ramamoorthy, F. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, 1976.
- [RSORS96] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, 1996.
- [Rus97] David A. Rusling. The linux kernel. <http://sunsite.unc.edu/LDP/LDP/tlk>, 1997.
- [RW85] S. Rapps and E Weyuker. Selecting software test data using data flow information. *IEEE Transactions of Software Engineering*, 11(4):367–375, 1985.
- [Saa91] Youssef G. Saab. Combinatorial optimization by stochastic evolution. *IEEE Transactions On Computer-Aided Design*, 10(4):525–535, April 1991.
- [Som92] Ian Sommerville. *Software Engineering*. Addison-Wesley, forth edition edition, 1992.
- [Spa90] E. H. Spafford. Extending mutation testing to find environmental bugs. *Software – Practice and Experience*, 20(2):181–189, February 1990.

BIBLIOGRAPHY

- [TM95] Ian Toyn and John A. McDermid. Cadiz – an architecture for z-tools and its implementation. *Software Practice and Experience*, 25(3):305–330, 1995.
- [Tov88] C. A. Tovey. Simulated simulated annealing. *Journal of Mathematical and Management Sciences*, 8:389–407, 1988.
- [vLA88] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: theory and applications*. Kluwer, 1988.
- [Yan95] Xile Yang. The automatic generation of software test data from z specifications. Technical report, Department of Computer Studies, University of Glamorgan, 1995.