# Searching for Test Data

Kamran Ghani

Submitted for the degree of Doctor of Philosophy

The University of York
Department of Computer Science

2009

# Abstract

In recent years metaheuristic search techniques have been applied successfully to solve many software engineering problems. One area in particular where these techniques have gained much attention is search based test data generation. Many techniques have been applied to generate test data for structural, functional as well as non-functional testing across different levels of abstractions of a software system. In this thesis we extend current search based approaches to cover stronger criteria, extend current work on higher level models to include further optimisation techniques, enhance current approaches to target "difficult" branches, and show how to apply current approaches to refine specifications generated automatically by tools such as Daikon.

# Contents

# Acknowledgment

I would like to thank my supervisor Professor John A. Clark. It would not have been possible for me to finish this work without his kind support, advice and encouragement throughout this period. I would also like to thank Professor Richard Paige for his insightful comments. Many thanks to Yuan Zhan for her kind support and collaboration on part of this work.

I would also like to thank all my friends and colleagues who always encouraged me to carry on this work.

Many thanks to my family who always stand by my side, my sisters, brothers and aunts who always prayed for my success, especially my elder brother Usman Ghani, who brought me up like a father. And thanks to my wife, who is spending some hard time without me.

# Declaration

The work presented in this thesis was carried out in the University of York from October 2005 to September 2009. Much of the work appeared in print as follows:

1. Kamran Ghani and John A. Clark *Strengthening Inferred Specification using Search Based Testing*, in proceedings of 1st International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2008, pp. 187-194, Lillehammer, Norway.

2. Kamran Ghani and John A. Clark *Widening the Goal Posts: Program Stretching to Aid Search Based Software Testing*, in proceedings of the 1st International Symposium on Search Based Software Engineering (SSBSE '09)

3. Kamran Ghani, John A. Clark and Yuan Zhan *Comparing Algorithms for Search-Based Test Data Generation of Matlab Simulink Models*, in proceedings of the 10th IEEE Congress on Evolutionary Computation (CEC '09), Trondheim, Norway.

4. Kamran Ghani and John A. Clark *Automatic Test Data Generation for Multiple Condition and MCDC Coverage*, in proceedings of the 4th International Conference on Software Engineering Advances (ICSEA '09b), Porto, Portugal.

For all publications, the primary author is Kamran Ghani with John A. Clark providing advice as supervisor. Tool support and advice was also given by Yuan Zhan for paper no 3 above. In all cases the actual experiments were designed and carried out by Ghani.

Chapter 1

# Introduction

## 1.1 Introduction

Dynamic testing — "the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour" [90] — is used to gain confidence in almost all developed software. Various static approaches, such as reviews, walkthroughs and inspections, can be used to gain further confidence but it is generally felt [22] that only dynamic testing can provide confidence in the correct functioning of the software in its intended environment.

We cannot perform exhaustive testing because the domain of program inputs is usually too large and also there are too many possible execution paths. Therefore, the software is tested using a suitably selected set of test cases. A variety of coverage criteria have been proposed to assess how effective test sets are likely to be. Historically, criteria exercising aspects of control flow, such as statement and branch coverage [136], have been the most common. Further criteria, such as data flow [154], or more sophisticated criteria such as MC/DC coverage [162] have been adopted for specific application domains. Many of these criteria are motivated by general principles (e.g. you cannot have much confidence in the correctness of a statement without exercising it); others target specific commonly occurring fault types (e.g. boundary value coverage).

Finding a set of test data to achieve identified coverage criteria is typically a

labour-intensive activity consuming a good part of the resources of the software development process. Automation of this process can greatly reduce the cost of testing and hence the overall cost of the system. Many automated test data generation techniques have been proposed by researchers. We can broadly classify these techniques into three categories: random, static and dynamic [128] [52].

Random approaches generate test input vectors with elements randomly chosen from appropriate domains. Input vectors are generated until some identified criterion has been satisfied. Random testing may be an effective means of gaining an adequate test set [56] but may simply fail to generate appropriate data in any reasonable time-frame for more complex software (or more sophisticated criteria) [46, 150].

With static techniques an enabling condition is typically generated that is satisfied by test data achieving the identified goal. For example, symbolic execution [97] can be used to extract an appropriate path traversal condition for an identified path. Such enabling conditions are solved by constraint solving techniques. However, despite much research, these approaches do not scale well, and are problematic for some important code elements, such as loops, arrays and pointers [117].

Recently Search Based Software Engineering (SBSE) [36, 77] has evolved as a major research field in the software engineering community. SBSE has been applied successfully to many software engineering activities ranging from requirements engineering to software maintenance and quality assessment. One major area where SBSE has seen intense activity is software testing [77]. McMinn [117] published an extensive survey in 2004. Since then there has been further activity in the search based software testing field. Active research is underway to improve the existing search based test data generation techniques and propose novel approaches to solve the test generation problem. However, despite much research, there are still limitations that have hampered the wide acceptance of these techniques. Also many areas are under-explored, and there are distinct possibilities for the successful use of search based approaches.

The work presented in this thesis aims to show that existing search based

testing techniques can be extended and shows new applications. We cover both code and higher-level model testing.

## 1.2 Hypothesis

The work in this dissertation is based on the following hypothesis:

> ***Search based test data generation techniques can be extended to satisfy criteria at both code and higher levels with increasing sophistication.***

To address the hypothesis, four topics are considered. Figure 1.1 summarises these.

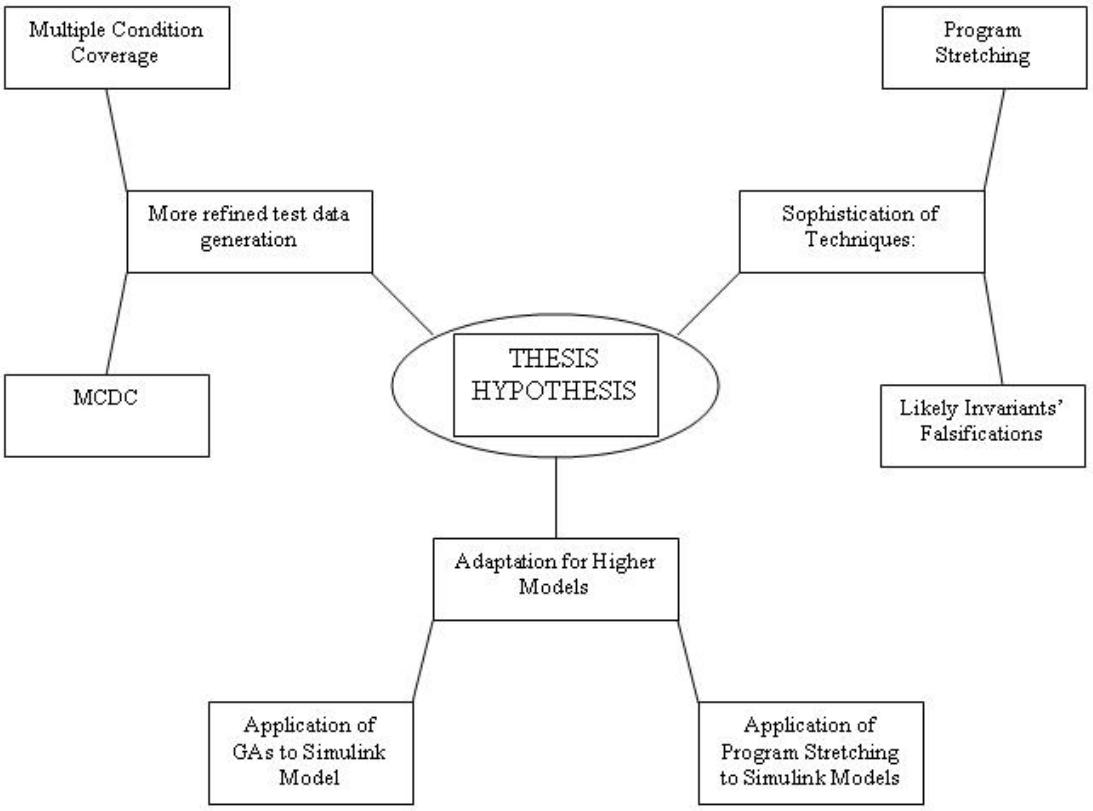


Figure 1.1: Thesis Hypothesis

## 1.3 Contributions

The contributions of this work are:

- Demonstration of a search based test data generation framework to generate test data for stronger coverage criteria.
- Extension of the existing work of Zhan and Clark [196] on test data generation of MATLAB Simulink Models.
- Investigation of program stretching—a novel approach to finding hard-to-get test data.
- Demonstration that search based test data generation techniques can be used to refine specifications inferred from dynamic trace data.

## 1.4 Outline of Thesis

The rest of this thesis is structured as follows.

**Chapter 2- Background**

Chapter 2 provides the background for this work. The chapter starts with a general introduction to software testing. A discussion of different coverage criteria at the code as well as the specification level is given. This is followed by a brief overview of search algorithms that have been proposed to generate input test data. Finally, a survey of the search based test data generation techniques is provided.

**Chapter 3-Structural Testing: Searching for Fine Grained Test Data**

This chapter introduces the framework for test data generation for modified condition decision coverage (MC/DC) and multiple condition coverage (MCC). All the important elements are presented with a discussion of how techniques are implemented to generate the required test data. This is followed by a discussion

of how the parameters for simulated annealing are tuned to conduct the experimental studies. Finally the results of experiments to evaluate the technique are presented and discussed.

**Chapter 4-Application of Genetic Algorithm to Simulink Models**

This chapter provides the work conducted to extend the current work of Zhan and Clark [196] on test data generation of Simulink models using search based techniques. A brief introduction to Simulink models is given followed by a discussion of how SBTDG techniques have been used to generate test data for it. A comparative study between SA and GA is presented.

**Chapter 5-Program Stretching**

In this chapter a program transformation based approach, 'program stretching', is introduced to generate test data for difficult branches. A survey of 'testability' transformation based approaches to overcome some of the limitations of search based testing is presented. This is followed by a proof of concept demonstration of how a 'program stretching' technique can be used to obtain test-data to cover 'difficult' branches (that is difficult for traditional non-state search based test data generation approaches).

**Chapter 6-Strengthening Inferred Specifications**

This chapter proposes the application of search based test data generation techniques to the refinement of automatically generated likely invariants and describes the development and use of a framework to do this.

**Chapter 7-Evaluation, Conclusion and Future Work**

Chapter 7 evaluates to what extent the hypothesis has been addressed. Conclusions are drawn and directions for the future work are given.

Chapter 2

# Literature Review

## 2.1 Software Testing

Software has become an intrinsic part of human life and it is important that it should perform its intended function. Otherwise it can cause frustration, loss of resources and even loss of life.

The main activity that attempts to prevent this and verify software quality and reliability is software testing. Testing is a dynamic activity, as it requires execution of program on some finite set of input data. Nevertheless there are other methods such as static analysis and formal proof of correctness. However, only testing can be used to gain confidence in the correct functioning of the software in its intended environment. We cannot perform exhaustive testing because the domain of program inputs is usually too large and there are too many possible input paths. Therefore, the software is tested against suitably selected test cases.

The main activities in software testing are test case generation, executing program using these generated test cases and evaluating the results. A test case is a set of test input data and the expected results. The test data is a set of input values to the program, which may be generated from the code or usually derived from program specifications. Program specifications also help in determining the expected results.

Program execution is the next important step. A test harness is often constructed by the tester to help in initializing global variables, if any, and to execute the

program with the test input data. The output of the program is then evaluated and decisions taken accordingly.

There are many techniques for software testing. We can broadly classify them in to two main categories, which usually complement each other: black box; and white box testing.

In black box testing techniques, tests are generated from informal or formal specifications of the software system. Techniques based on informal specifications include Boundary Value Analysis (BVA), Category Partition (CP), Classification Tree Method (CTM), Cause-effect graphs, Equivalence Partitioning (EP) and random testing etc. Model-based testing which includes test case generation from formal models such as Z, B or VDM specification and graphical models such as Finite State Machines (FSM) and statecharts, are techniques based on formal specifications. Recently semi formal specifications such as UML models have also been used for software testing.

White box techniques exercise elements of the programs on a given set of data to observe the behaviour of the software. Techniques such as data-flow testing, domain testing, mutation testing and path testing are some examples in this category.

As stated earlier, exhaustive testing i.e., using all inputs values for testing, is usually computationally impractical. Instead we usually select a small subset of input data. It is important that this subset is good enough to detect many of the errors. The notion of an adequacy criterion, the subject of next section, is usually a measure which gives us an insight about the goodness of the data, i.e., how thoroughly does it test the software.

An adequacy criterion is decided which is to be satisfied by a test set. Generation of a test set then follows. Many approaches can be used for test set, or more specifically, test data generation. These can be classified into two categories: static; and dynamic.

In static approaches such as symbolic execution, a model of the source code is formed and tests are generated from it. The code itself is not executed.

Some of the problems experienced by static approaches are overcome by dy-

namic approaches. Such approaches involve the execution of the actual code elements. In such techniques, information obtained from such execution is exploited for test data generation. Heuristic optimization based techniques are among such approaches where test data generation is modelled as a search problem and then some heuristic is used which guides the search to find the test data. Section 2.6 provides an overview of these techniques.

In the context of this dissertation, we consider a program $P$ with an input vector $v = v_1, v_2, ..., v_n$ belonging to an input domain or search space $S$, where $v_i \in S_{v_i}$ and $S = S_{v_1} \times S_{v_2} \times ... \times S_{v_n}$. A test case $T_i$ is any $v$, used to execute the $P$ for the purpose of testing. A test set $T = T_1, T_2, ..., T_n$ is a collection of test cases.

## 2.2 Adequacy Criteria

When performing testing, two important questions usually confront a software tester: when should testing be stopped and how do we decide that the test cases constituting a test set are adequate for achieving the stated objective. The answer to the first question is given by test adequacy criteria. Whereas the test case selection criteria give an answer to the second. "However, the fundamental concept underlying both test case selection criteria and test data adequacy criteria is the same, which is the notion of test adequacy" [199].

Another term associated with testing is *test coverage*. Software test coverage is "any metric of completeness with respect to a test selection criterion" [22]. It is usually measured by calculating how thoroughly a program is exercised by a given test suite. In other words, coverage can be used to measure the extent to which an adequacy criterion is satisfied. Therefore coverage criteria are a type of adequacy criterion that specify the percentage of requirements that must be covered [122]. However most of the time the two terms i.e., coverage criteria and adequacy criteria are used synonymously and we will consider it this way, unless otherwise specified.

This section discusses the most commonly used software testing coverage cri-

teria found in the literature. A brief taxonomy of coverage criteria is given, which is followed by an overview of these criteria.

Coverage measures may be classified in to two main categories: structure-based; and fault-based coverage criteria. A structure-based coverage criterion requires the execution of particular components of a program, whilst fault-based criteria are based on the measurement of ability of test suites designed to demonstrate the absence of a set of pre-specified faults in the software.

## 2.3 Structure Based Coverage Criteria

These criteria are in turn are divided into program-based and specification-based coverage criteria. We begin with program based criteria.

These specify testing requirements in terms of the coverage of a particular set of elements in the structure of the program and include control-flow based and data-flow based criteria.

### 2.3.1 Control Flow Based Testing

The simplest control-flow criteria are known from the 1960s and 70s. The descriptions that follow are based on the well-known book by Myers [136].

#### Statement Coverage ($SC$)

Every reachable statement $S_r$ in a program $P$ has been executed at least once. It is possible that some number $S_i$ of statements lie on infeasible paths, in which case they cannot be executed. Thus statement coverage can be defined as

$SC = S_c/(S_t - S_i)$, where $S_c$ is the number of statements covered and $S_t$ is the total number of statements in $P$. Statement coverage is achieved when $SC = 1$.

#### Decision Coverage ($DC$)

$DC$ can be stated as: every reachable decision $D_r$ in $P$ has taken all the possible outcomes at least once. $DC$ can be defined as $DC = D_c/(D_t - D_i)$, where $D_c$ is

the total number of decisions covered, $D_t$ is the total number of decisions and $D_i$ is the number of unreachable decisions.

$DC$ is popularly known as branch coverage, and may be stated as coverage of all reachable edges or branches in the flow graph.

#### Condition Coverage ($CC$)

This criterion can be stated as: every statement in the program has been executed at least once, and every condition in each reachable decision has taken all possible outcomes at least once. Thus $CC$ can be defined as

$CC = C_c/(C_t - C_i)$, where $C_c$ is the number of conditions covered, $C_t$ is the total number of conditions and $C_i$ is the number of unreachable conditions.

#### Decision/Condition Coverage ($D/CC$)

Every statement in the program has been executed at least once, every decision in the program has taken all possible outcomes at least once, and every condition in each decision has taken all possible outcomes at least once. It can be defined as $(D/CC = (D_c + C_c)/((D_t - D_i) + (C_t - C_i))$ [113]

#### Multiple Condition Coverage (MCC)

Every statement in the program has been executed at least once, and all possible combinations $C_p$ of condition outcomes in each decision $D$ have been invoked at least once. This criterion is also called as Extended Branch Coverage Criterion [199].

In the control flow based coverage criteria subsumption hierarchy as shown in Figure 2.1 [32], multiple condition coverage is the strongest criterion. It requires test cases that cover all the conditions in a decision. For example, consider the truth table in Table 2.1. For a decision containing two conditions as $C_1 \wedge C_2$, we need test cases to exercise all 'true' and 'false' combination of $C_1$ and $C_2$ i.e., $TT$, $TF$, $FT$, and $FF$. In general, if a *decision* $D$ contains $n$ *conditions* $C$, we require atleast $2^n$ test cases to satisfy multiple condition coverage.

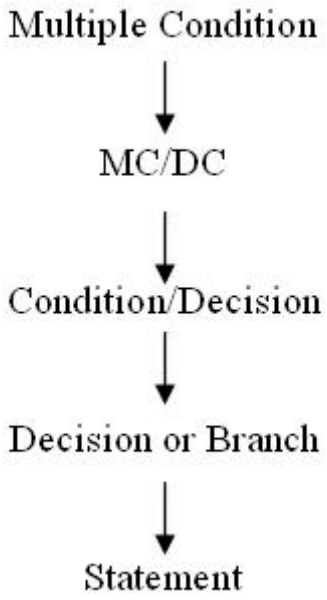


Figure 2.1: Control Flow Subsumption Hierarchy

| Test Case No | $C_1 \wedge C_2$ | outcome |
|---|---|---|
| 1 | TT | T |
| 2 | TF | F |
| 3 | FT | F |
| 4 | FF | F |

Table 2.1: Required combination of test case sequences for multiple condition coverage.

### Modified Condition/Decision Coverage (MC/DC)

The number of tests required to satisfy multiple decision coverage increases exponentially with the number of conditions, it can become very expensive for decisions with large numbers of conditions. There may also be infeasible combinations of conditions. For example consider the 2nd branch in Figure A, appendix A. With given conditions, the combinations $TTT$, $TTF$, $TFT$ and $FTT$ are all infeasible. Filtering out such combinations further increases the cost of this criterion and hence it may not be practical to apply it for large and complex systems.

MC/DC on the other hand is a more practical criterion and hence usually a testing requirement for critical systems such as those developed in the avionics

| Test case No | $C_1 \wedge C_2$ | outcome |
|---|---|---|
| 1 | TT | T |
| 2 | TF | F |
| 3 | FT | F |

Table 2.2: Required combination of test case sequences for MC/DC.

domain [162]. It is satisfied when (i) every condition in a decision in a program has taken all possible outcomes at least once, and (ii) each condition has been shown to independently affect the decision's outcome. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions. Consider Table 2.2. Requirement (i) is satisfied by test cases 2 and 3 for condition $C_1$ and by test cases 1 and 2 for condition $C_2$. Requirement (ii) is satisfied by test cases 1 and 2 for $C_1$ and by 1 and 3 for $C_2$. We can satisfy MC/DC for a decision with a minimal set of $n + 1$ test cases, where $n$ is the number of conditions in the decision. Considering the second branch in Figure A in Appendix A, the minimal set also eliminated the infeasible combinations as were incurred in multiple condition coverage. MC/DC coverage has gained significant exposure in real world software testing since it is a mandatory requirement of testing of software of high integrity in the civil avionics sector [162].

Another criterion, which is somewhat related to the above and based on control flow is path coverage.

### Path Coverage

Path coverage usually covers the requirement of checking of all the combinations of branches. The path coverage criterion is very strong but may not be practical as there may be infinite number of paths. But in practice a finite number of paths, which must include the most important sub paths, are chosen based on some criteria [149, 116].

Along with the above-mentioned criteria, we also find many other criteria like relational coverage [112] etc. A long list can be found in [95]. However, most of these have little value on the grounds and are subsumed by stronger and more

widely accepted criteria.

### 2.3.2 Data Flow Based Criteria

In data flow based coverage criteria it is attempted to examine the association between the definition of a variable and its uses in the program. The exercising of a definition-use association can be viewed as requiring traversal of a selected sub path, which originates at the definition, terminates at the use, and is responsible for establishing the definition-use association. The following terminology based on each variable occurrence is necessary for understanding these criteria:

#### Variable Definitions

When a variable is assigned to or contained in an input statement that variable is said to undergo $definition$. For example, a statement such as x=10 in a program is a $definition$ of a variable $x$.

#### Def-Use

An occurrence of $x$ in a statement like $y = x + 1$ is termed a $use$ of $x$. A $use$ is either a $c - use$ or a $p - use$. The $definition$ and $use$ occurrences of a variable are termed as $def - use$ pairs.

#### c-use or computation-use

A $computational - use$ occur when the variable forms part of the right hand side of an assignment statement or is used as an index of an array or contained in an output statement. For example, $y = x + 1$ gives rise to a $computational - use$ of x.

#### p-use or predicate use

A predicate use occur when a variable appears in a predicate in a conditional branch statement, e.g., the use of $x$ in the boolean expression $if(x <= n)$.

### Definition-clear path

A path $(i, n_1, n_2, ..., n_m, j)$ is a $definition - clear$ path, with respect to a variable $x$, from the exit of $i$ to entry of $j$ if $n_1$ through $n_m$ do not contain a $definition$ of variable $x$.

### Loop-free path

A path is $loop - free$ if all visited nodes are distinct.

### Du-Path

A path $(i, n_1, n_2, ..., n_m, j)$, is a $du - path$ with respect to a variable $x$ if $i$ has a definition of $x$, and either j has a $c - use$ of $x$ and the path $((i, n_1, n_2, ..., n_{m,j})$ is $def - clear$ path with respect to $x$ or $(n_m, j)$ has a $p - use$ of $x$ and the path $(i, n_1, n_2, ..., n_m)$ is both $def - clear$ path and $loop - free$ path with respect to $x$.

Following are the data flow based coverage criteria proposed by Rapps and Weyuker[154]:

### All-Def Criterion

The $all - def$ or $all - definitions$ criterion requires that an adequate test set should cover, at least once, all $definition$ occurrences of a variable $x$ in the sense that, for each definition occurrence, the testing paths should cover a path through which the $definition$ reaches a $use$ of the $definition$.

### All-uses Criterion

The $all - uses$ criterion requires that all of the uses of a variable $x$, which includes the $def - use$ as well $p$ and $c - use$ should be exercised by testing at least once. This criterion subsumes the $all - def$ criterion.

Similarly based on the use occurrences, i.e., either the predicate or computational, the following criteria were proposed by Rapps and Weyuker[154]

**p-uses**

requires only $p-uses$ of the variable $x$ be exercised.

**c-uses**

requires only $c-uses$ of the variable $x$ be exercised.

**All c-uses/some p-uses criterion**

requires that all of the $c-uses$ and at least one $p-use$ of a variable $x$ be exercised.

**All p-uses/some c-uses criterion**

requires that all of the $p-uses$ and at least one $c-use$ of a variable $x$ be exercised.

**All-Du-Paths Criterion**

An enhancement to the above criteria has been given by Frankle et al. [63] and Clarke et al. [37] in the form of $All-DU-Paths$ Criterion, which requires that every $du-path$ from each variable $definition$ to every $use$ of the $definition$ be executed at least once. It subsumes the $all-uses$ criterion. However, all the above criteria have a number of limitations [199] that affect their usage for testing. These criteria are extended to overcome such limitations, like in the form of intraprocedural (unit testing) and interprocedural testing techniques (integration testing) [71].

## 2.4 Specification Based Coverage Criteria

A specification specifies the properties, i.e., the behaviour, that the software must satisfy. Since the main purpose of testing is to ensure that the software performs what it is suppose to do and reveal any abnormal behaviour, specifications play an important role in software testing. There are two roles that a specification can play in software testing [199]: to act as a platform for the test oracle [160]; and to act as a base for test case selection and hence measure test adequacy. The

advantage [13, 10] of using specification is that it allows test creation earlier in the software development process and helps to catch the errors before they are too expensive to correct. Test case generation in the early stages may also reveal inconsistencies and ambiguities in the specification. Similarly, implementation independent essential test data, output checking and conformance testing are all based on specification.

There are many approaches to specification based testing [144], i.e., Formal Specification based approaches, Algebraic approaches and State-based approaches. There are also semi-formal specifications such as UML. These have also been used in testing. In the following section Formal Specification based, Semi Formal Specification based and State-based coverage approaches are briefly surveyed.

### 2.4.1 Formal Specification Based Approaches

Formal specification based approaches use mathematical languages such as B, Z and VDM for the generation of test data. There is considerable literature on automatic and semi-automatic test case generation from formal specifications. See for example [169], [41], [53], [30], [88], [104], [85]. The method developed by Dick and Faivre [53] has gained significant popularity. This is based on reducing the specification of operations to Disjunctive Normal Form (DNF). The DNF is converted into Finite State Machines (FSM), which is used for deriving test suites. The original method was applied to VDM, but has also been applied to Z [88, 172] and to B [179].

Since FSMs or statecharts are represented as directed graphs, the program based coverage criteria can be applied in order to measure the coverage of the testing suites based on formal specification languages [199].

Ammann and Offutt [17] applied a form of category partition method to the Z specification to derive a combination of choices to be tested. They proposed three coverage criteria, which are given below.

#### All Combination Criterion

This criterion requires that software be tested on all combinations of choices. Since there is very large number of possible combinations of choices, this criterion is not efficient.

#### Each-choice-used Criterion

This criterion requires that the test set $T$ incorporate each choice at least once. However, this criterion was also considered ineffective as it can lead to undesirable test sets because we may be able to select 'unused' choices, which are never used in the normal mode of operation of a system.

#### Base-Choice-Coverage Criterion

Ammann and Offutt [17] advocated this criterion as the minimum adequate criterion. They argued that each system has a normal mode of operation and that normal mode corresponds to a particular choice in each category. This particular choice is called as *basechoice*. Thus $base - choice - coverage$ criterion requires that each choice in a category be tested by combining it with the base choice for all other categories. This causes each non-base choice to be used at least once, and the base choices to be used several times.

### 2.4.2 State Based Approaches

State-based specification describes software in terms of state transitions. State-based specification consists of preconditions defined on transitions, which are values that specification variables must have for the transition to be enabled, and triggering events, which are changes in variable values that cause the transition to be taken [13]. The state machines that are usually used for testing are Finite State Machines (FSM), Extended Finite State Machine, Abstract State Machine and statecharts.

State-based testing can be dated back to the late 70's when Chow [35] proposed the W-method for finite state machine. However, only in late 1990s was there some

significant attention to this approach for testing software, e.g., [144],[13], [10],[25], [87]. Many coverage criteria were proposed which are given as follows.

Offutt [13] introduced several criteria for system level testing. These criteria include *Transition Coverage (TC)*, *Full Predicate Coverage(FP)*, *Transition Pair Coverage (TP)* and *Complete Sequence (CS)*. To apply these, a state-based specification is viewed as a directed graph, called the specification graph. Each node represents a state (or mode) in the specification, and edges represent possible transitions.

**Transition Coverage Criterion (TC)**

This criterion requires that every precondition in the specification be tested at least once. In terms of Specification graph it requires that each transition is taken at least once and can be stated as: "the test set $T$ must satisfy every transition in the $SG$," where $SG$ stands for the specification graph.

**Full Predicate Coverage Criterion (FP)**

This criterion requires that each clause in each predicate on each transition is tested independently at least once. Formally this criterion can be stated as: "for each predicate $P$ on each transition, test set $T$ must include tests that cause each clause $c$ in $P$ to result in a pair of outcomes where the value of $P$ is directly correlated with the value of $c$" .

**Transition-Pair Coverage Criterion (TP)**

This criterion requires that pairs of adjacent transition be tested. This criterion can be stated as "for each pair of adjacent transitions $Si : Sj$ and $Sj : Sk$ in Specification Graph $SG$, test set $T$ must contain a test that traverse the pair of transition in sequence."

#### Complete Sequence Criterion

A complete sequence is a sequence of state transitions that form a complete practical use of the system. This criterion states "test set $T$ must contain tests that traverses 'meaningful sequences' of the transitions on the $SG$, where these sequences are chosen by the test engineer based on experience, domain knowledge, and other human-based knowledge".

All the above criteria are mostly motivated by control flow criteria but are applied to the specification instead of code. TC is similar to branch coverage and FP is analogous to Modified Condition/Decision Coverage (MC/DC). FP coverage tests transition independently, but does not test sequence of state transitions. TP takes this into account and thus tries to identify faults that may arise when invalid sequence of transitions is allowed, or a valid sequence of transition is not allowed. Similarly the Complete Sequence criterion can be compared to 'selected path coverage'. In literature we also find Abstract State Machine (ASM) based coverage criteria. Gargantini and Riccobene [12] proposed ASM-based criteria as rule coverage, rule update coverage, parallel rule coverage, strong parallel rule coverage and MC/DC.

### 2.4.3 UML Based Coverage Criteria

Currently there has been growing interest in using UML for software testing. Many techniques and criteria have been proposed based on UML diagrams and supporting specification documents. UML has been extended in the form of the UML 2 Testing profile [7] in order to provide a mean for using UML as test specifications. Many coverage criteria can be found based on UML diagrams. A comprehensive survey of these criteria is presented by McQuillan [122]. A brief account of these criteria is given in the following sections.

#### Use case based criteria

Use cases are used to model the behaviour of the system from the user's point of view. In UML based development they are considered the primary documents for

capturing requirements and validating the system.

Use cases are diagrammatically represented in the form of use case diagrams. A use case diagram usually represents all the use cases of the system. The important elements of the use case diagram are the *actor*, the *use case*, and the *relationships*.

As stated earlier, the main purpose of use cases is to describe the behaviour of the system and provide a good source of information for testing. Use cases can be used in two ways for testing depending upon the source of information: either using the use case textual description or the use case diagram.

Testing based on the use case description is mostly used for exercising identified scenarios. Heumann [83] gives a three-step approach for generating test cases from a fully detailed use case. In this approach, first a full set of scenarios is generated for each use case from its textual description. Then for each scenario at least one test case is identified along with the necessary conditions that will execute it. In the final step data values are identified for each of the test cases. The main drawback of this approach is that the whole process is manual, which is laborious and prone to error. Also the use cases describe high-level system functionality and hence the test cases generated also reflect that level. It may not be of much help if we want to test low-level details of the system.

Briand and Labiche [28] proposed an approach called TOTEM (Testing Object Oriented systEm with Unified Modelling Language) for system testing. In this approach use cases are used to derive the corresponding sequence, collaboration and class diagrams. Requirements for system testing are then derived from these artefacts, which are then transformed into test cases, test oracles and test drivers. Ryser and Glinz [163] also proposed an approach for testing based on use cases. In this approach use case description is used to construct state charts from which test cases are then generated.

As far as coverage criteria are concerned, to the best of our knowledge, no work could be found based on use cases textual description. This is because of the non structured nature of the document. However, they can be useful for scenario based testing. A number of criteria can be found for use case diagrams. Reuys [158] pointed out a number of use case diagram based criteria citing Winters

[188]. These criteria are Use Case Step Coverage, Use Case Branch Coverage, Use Case Scenario Coverage, Use Case Boundary Body Coverage and Use Case Path Coverage. However, no detailed work could be found on this. Based on Binder [24], Hans [69] also proposed a number of distinct coverage criteria, similar to the traditional test coverage measures that are based on program flow-graphs. These criteria give coverage of all the nodes and arrows in a use case diagram. The criteria are Coverage of at least one Use Case, Coverage of at least every Actor's Use Case and Coverage of at least Every Fully Expanded Inclusion, Extension and Uses Combination.

The only defined criteria based on use case diagrams are given by [29]. Two classes of criteria are proposed. The first based on relationship and the second based on the combination of the extended relationship. One criterion from each class is then described. These criteria are similar to that proposed by [69]. These criteria are All-Association-Inclusions-Extensions Criterion ($C1$) and the All-extended-combinations Criterion ($C2$) respectively.

$C1$ requires that for a test set $T$ and a use case diagram $D$, $T$ must cause each association, include and extend relationship in $D$ to be exercised at least once. Thus this criterion considers the coverage of all type of relationship in a use case.

$C2$ takes into account the conditions in the extend relationship that must be satisfied if the extension is to take place. This criterion requires that for a given test set $T$ and a use case diagram $D$, for each use case extended by at least two other use cases, $T$ must cause all the combinations of exercising and non-exercising the extend relationships to be exercised at least once.

To support the application of the criteria, a testing tool called as UCT (Use Case Tester) was also developed and a case study conducted on a small application, which revealed some scenarios not identified by the functional tests developed based on Heumann's [83] approach.

However, the above criteria are still immature and testing tools could not achieve the complete coverage for a small case study. Besides the criteria only determine user functionality, but not how to test the functionality itself. The effectiveness of these criteria cannot be assessed and what value do they hold to

prevent faults creeping into the implemented system.

### Criteria based on Interaction diagrams

The two kinds of interaction diagrams in UML are communication diagrams and sequence diagrams. A communication diagram, previously called a collaboration diagram, shows collaboration among objects to achieve a behavioral goal. Sequence diagrams are very similar and also show interaction among objects but emphasize the time ordering of messages that are passed between objects. Many coverage criteria have been proposed for interaction diagrams. Abdurazik and Offutt [11] adapted the *all definition-uses* criterion in the context of UML communication diagrams [167]. They proposed the *'message sequence path'* criterion. A *message sequence path* is an ordered sequence of all messages in a communication diagram. According to this criterion, a test set must contain tests for each sequence path in a communication diagram [122]. Andrews et al. [18] proposed many criteria for communication diagram. The *condition coverage (Cond)* criterion is similar to branch coverage and requires each condition in the communication diagram to take both *true* and *false* values. When a condition contains more than one clause then *full predicate coverage (FP)* needs to be satisfied, which requires every condition to take both *true* and *false* values while all other clauses in the predicate have values such that the value of the predicate is the same as the value of the clause being tested. Another criterion defined is *each message on link (EML)*, which requires that test cases must execute each message on a link connecting two objects at least once in a communication diagram. They also defined the *all message paths (AMP)* criterion, which is similar to *all message sequence paths* criterion described earlier. *Cond* and *FP* criteria target faults related to using inadequate conditional control flow structure while EML and AMP target faults related to interactions between objects [18]. Wu et al. [190] describe criteria which require each transition and each valid sequence in each collaboration diagram to be tested at least once.

Rountev et al. [161] suggested three coverage criteria based on the sequence diagram. These criteria are generalizations of traditional control-flow criteria,

such as branch and path coverage. They define the *interprocedural restricted control-flow graph*(IRCFG), which represents the set of message sequences in a sequence diagram. An IRCFG contains *restricted CFGs* (RCFGs), each corresponds to a particular method and similar to the CFG of that method but restricted to the flow of control relevant to the message sending. The IRCFG is used to define a set of coverage criteria. The *All-IRCFG-Paths* criterion requires coverage of all paths in the IRCFG. It is similar to path coverage for the traditional control flow graph. Like path coverage, it is possible that the number of start-to-end message paths can be very large, thus making this criterion infeasible to achieve [122]. Therefore, Rountev et al. [161] proposed the *All-RCFG-Branches* criterion, which requires coverage of all RCFG edges. This criterion is weaker but more achievable. They further proposed a 'more weaker but easier to achieve' criterion termed as *All-Unique-Branches*. Since the same RCFG edge can appear many times in IRCFG, this criterion requires at least one occurrence of each RCFG to be covered. In practice these criteria are not easily achieved because of the number of paths in an IRCFG. The authors themselves concluded that *All-IRCFG-Paths* is impractical due to very large start-to-end paths even in a low depth IRCFG. The other criteria also require substantial efforts to be achieved. Similar criteria are also proposed by other researchers [24, 28] where paths are derived from sequence diagrams and then tests are sought to traverse those paths.

### Criteria based on Class diagrams

A class diagram shows the classes of the system, their relationships (association, aggregation, inheritance) and the operations and attributes of the classes [16]. Andrews et al. [18] proposed three criteria based on the form of constraints (association-end-multiplicities, generalisation and OCL statements) present in the class diagram. The *Association-end-multiplicity* (AEM) criterion requires tests that "exercise configurations that contain boundary and non-boundary occurrences of links between objects". The *Generalisation*(GN) criterion target faults caused by inheritance relationships. This criterion requires tests that create each generalisation/specialisation at least once. The third criterion defined by Andrews

et al. is the *Class Attribute*(CA) criterion. This criterion targets faults that occur because of violation of *value spaces* of attributes that have been restricted by OCL constraints. Using the category partition method, possible values for each attribute in a class are produced. The Cartesian product of these values with the other attributes' values is taken and a valid aggregated set of attribute value combinations is identified. CA requires coverage of this set by tests for each class.

#### Criteria based on other UML diagrams

Many criteria are based on other UML diagrams such as state machines and activity diagrams. Since UML state machines are specialised FSMs therefore, the same criteria as described in section 2.4.2 have been proposed (such as those proposed by Offutt and Abdurazik [146]). Activity diagrams are special cases of statechart diagrams and so, the same criteria can also be applied to activity diagrams [122].

## 2.5 Fault Based Adequacy Criteria

Fault based testing selects tests that would identify whether specific faults are present in the software. Faults are often injected into the software and tests developed to detect those injected faults. (Error seeding and mutation testing fall into this category.) Alternatively, a hypothetical fault may be used to suggest tests; e.g., boundary value errors.

Many techniques have been developed in this regard which include error seeding [131] [99], mutation testing [159], and its variants [89] [189] [49] [48] [135] and perturbation testing [194]. An adequacy criteria for fault based testing is measured in terms of its ability to detect the injected fault. Following are the brief description of fault based testing techniques and their adequacy criteria.

### 2.5.1 Error Seeding

Error seeding is among the earliest fault based testing techniques [181]. It can be defined as, "the process of intentionally adding known faults to those already

in a computer program for the purpose of monitoring the rate of detection and removal, and estimating the number of faults remaining in the program" [139]. In error seeding the errors are deliberately introduced into the program and the program is then tested to find the actual (non-seeded) errors and the seeded errors. An estimate of the actual errors $N$ remaining in the program is then calculated from the total number $F$ of seeded errors and the number of actual $n$ and seeded $f$ errors that are found during testing from the following relation

$(F - f)/F = (N - n)/N$.

Other forms of the relationship are also given as indicated by [181, 199].

An advantage of the error seeding technique is that we can easily measure the coverage of the testing process. This also gives us a measure of the testing quality. If $f/F$ is small then this means that testing quality must be poor [199].

Error seeding is a simple concept and provides a stopping condition for testing, but it also has many drawbacks, which questions its effectiveness in revealing the actual software error. It is based on the hypothesis that the proportion of seeded errors found by the test process is the same as that of the actual (non-seeded) errors. In other words the seeded faults must be the representative of actual faults. But in reality it is not the case. Errors introduced deliberately by conscious efforts are not likely to produce the same faults in the system as those introduced by the unconscious process [99]. The introduction of errors is also an issue. Usually faults are planted manually which is a laborious and time consuming activity and hence cannot be applied for large projects.

### 2.5.2 Mutation Testing

In order to overcome the weaknesses in error seeding, the concept of Mutation testing was introduced. The initial concept was proposed by Richard Lipton in 1971 in a class term paper titled 'Fault Diagnosis of Computer Programs' [145]. However [159] is considered the primary reference that explains the method. Unlike error seeding in which errors are planted in a single program, mutation testing uses variants of the main program called mutants. The mutants differ from the main program in a single small way, for example, replacing an instance

of $>$ with an instance of $<$ or a $+, *, /$ for a $-$. Each mutant is then executed on a selected set of test cases. Testing is stopped when a mutant produces a different output on a specific test set than the original program or the test set is exhausted. The mutant is said to be killed in the former case and alive in the latter. A mutant may live because either the test set is not adequate or the mutant is semantically equivalent. An equivalent mutant is the one that produces the same output as the original program, no matter on whatever test data it is exercised, thus distinguishing these cases is very hard.

The number of dead and live mutants also gives us an adequacy criteria for mutation testing usually called mutation adequacy or mutation score and is defined by the following relationship;

$M = K/Z$,

Where,

$M$=Mutation score or Mutation adequacy.

$K$= Number of killed mutants.

$Z$=Total number of non-equivalent mutants.

Where as non-equivalent mutants is the difference of total number of mutants and equivalent mutants.

Mutation testing is based on two assumptions, i.e., the competent programmer hypothesis and the coupling effect.

The competent programmer hypothesis assumes that the programmer is competent enough and is capable of producing the programs that may slightly deviate from the correct program. Therefore mutants are created reflecting this variation. The second assumption is that of coupling effect which define relationship between simple and complex faults. Thus according to this hypothesis, a mutation adequate test set capable of discovering simple mutants from the original, will also be capable of revealing more sophisticated and complex faults.

Mutants are generated by applying the mutation operators to the original program. A mutation operator defines a simple transformation rule like replacing $+$ with $-$. These operators are designed on the basis of errors that programmers usually made in writing programs. We can find mutation operators for most of

the commonly used high level languages [96, 73].

Mutation testing has many strong points. It allows a greater degree of automation. Mutant generation by the application of mutation operators, compilation and execution of mutants as well as comparison of results all can be automated. Many tools have been developed in this respect starting from PIMS in early 70s [145]. Similarly the Mothra mutation toolset [33] was developed in 1980s, which is one of the most widely known mutation system. Recently we find many mutation testing tools for example, Jave [109] and Jester [134] for Java.

There are also many issues associated with mutation testing. This technique is based on assumptions as described earlier. These assumptions create doubts in one's mind about mutation reliability as these assumptions may not hold. However, empirical and theoretical studies show the validity of these assumptions. For example [19] conducted empirical studies which supports the 'competent programmer' assumption. Whereas [141] conducted empirical studies on coupling effect assumption. These studies reveals that the assumption do hold to a great extent. Wah [182] attempted to give a theoretical explanation of the coupling effect.

Equivalent mutants are also undesirable in mutation. Equivalence of mutant itself may be undecidable. However, using automatic detection techniques Offutt [142] considerably reduces the number of equivalent mutants.

Another main issue which hampered the practical use of mutation is the requirement of large computational resources for generating and executing the large number of mutants. Estimation shows that an n-line program will produce mutants of the order $n^2$ [199] referring [89]. Other studies [140] shows that the number of mutants generated for a program is roughly proportional to the product of the number of data references and the number of data objects. This gives an idea of how much computation resources are required for a fairly moderate program. In addition to this a significant manual effort is required in this technique for analysing tests results and examining equivalent mutants.

In order to deal with the above issues many strategies have been used which resulted into many variants of the mutation. Offutt and Untch [145] classified

these strategies into three types, i.e. do fewer, do smarter or do faster.

Do fewer approaches attempts to execute fewer mutants without significant information loss. These approaches include Selective Mutation [143] and Mutant Sampling. In Selective Mutation, only those mutants are selected which are truly different from others. This is achieved by using effective mutation operators only, which are the operators that generates non-redundant mutants. Whereas in Mutant Sampling instead of using a complete mutation set only a subset is used. Random sampling and Bayesian sequential probability techniques have been applied [145] and show that we can achieve high performance by using only a subset of complete set.

Do Smarter approaches, though, attempts to execute the same set of mutants as that of strong mutation but in a smarter way. These approaches include Weak Mutation [89] and Firm Mutation [189] techniques. In weak mutation the internal states of the mutant and original program is compared immediately after the execution of the mutated portion of the program. If the states differ the mutant is said to be killed. It is possible though that the external state after complete execution may be the same as that of the strong mutation. So an adequate test set for weak mutation may not be so for strong mutation. [189] gave an intermediate approach, i.e., weaker than strong mutation but stronger than weak mutation. This technique suggests that instead of making the comparison at some specific stages it can be made at any stage in the program. However, this approach does not explain how to select such areas systematically. Along with these approaches there are many other do smarter approaches for example, adapting the mutation analysis system to a specific processor [114, 101, 141, 34].

Do faster approaches tried to reduce the execution time. Schema-based mutation analysis [178], in which all mutants are encoded into one metaprogram and then compiled and run, is one example of such approaches. This technique shows a performance improvement of 300 percent.

Regardless of the approach one employs, it is often much more expensive to obtain a high mutation score than to satisfy other criteria, for example, achieving 100 percent decision coverage.

### 2.5.3 Conclusion

Coverage refers to the extent to which a given verification activity has satisfied its objectives. It is a measure not a method or a test. We have got coverage criteria to get the coverage of a testing process. Since it is not possible to test software completely, so we choose software process that give 'certifiable' coverage. To get that coverage we use a suitable criterion. Achieving 100 percent coverage by a criterion does not employ that we have tested the software completely. However, each of the different coverage criteria attempts to capture some of the important aspects of a program structure, for example, achieving 100 percent statement coverage increases our confidence that we have revealed all the error that may occur due to the execution of statements in a program.

## 2.6 Search Based Optimsation Techniques

Optimization problems are very common in many fields and especially in engineering. To solve these problems many techniques have been developed. In an optimisation problem we seek to find a maximum or minimum value of a function. Generally, an optimisation problem can be stated as [155, 176]

$Minimize\ f(x)$

Subject to $g_i(x) = 0$, $i$=1,2,...,$n$

where $x$ represents a vector of decision variables, i.e., the variable that require the assignments of values, f() represents functions such that if $X$ represents all feasible solutions then $f : X \rightarrow \Re$ and $g_i()$ represents a constraint that limits the acceptable values of $x$. The objective is to find

$x^* \in X$ such that $f(x^*) \leq f(x) \forall x \in X$.

We shall generally be concerned with minimization, but, of course, there is an equivalent formulation for maximization problems. $f()$ is usually called the objective, cost or fitness function. Cost is generally minimized; fitness is generally maximised.

In this section we consider various optimisation techniques that have been utilized to generate input test data generation.

Select an initial solution $S_0$;
Repeat

Generate a move $S' \in N_{S_0}$;

Where $N_{S_0}$ defines the neighbourhood of $S_0$

$If\ f_{(S')} > f_{(S_0)}$
$S_0 = S'$

Until $S_f$ or max allowed time is reached

Table 2.3: Hill-Climbing algorithm

### 2.6.1 Hill-Climbing

Hill Climbing is a simple optimisation technique which resembles 'climbing' up a hill to reach the 'top'. The algorithm starts by selecting an initial candidate solution $S_0$ as the current solution. (Often this is chosen randomly.) Another candidate solution $S'$ is chosen in the neighbourhood of $S_0$ . If $S'$ is better than $S_0$ then it is chosen, otherwise it is discarded and the neighbourhood of $S_0$ is searched again for a better solution. The process continues until no further improvement in the current solution $S_f$ can be made. At this time the algorithm is considered to have reached a local optimum. The algorithm is given in Table 2.3.

### 2.6.2 Simulated Annealing

Simulated annealing is a form of neighbourhood search which mimic the process of annealing in metals. The original algorithm was given by Metropolis et al in 1953 [125]. It was 30 years later that Kirkpatrick [98] suggested its application to the optimisation problems. Since then SA has been applied successfully to solve many optimisation problems.

This approach can be taken as a variant of the Hill-Climbing. The problem with Hill-Climbing is that it can get stuck at a local optimum. Many improvements have been suggested such as starting the algorithm from different points or

increasing the neighbourhood size etc, but none of these approaches have proved satisfactory. SA solves the problem to some extent by allowing worsening moves to be accepted with some probability, as shown below;

$P = e^{-\delta/t}$.....(1)

Where P= Probability of accepting the move.

$\delta$=Change in the Cost function.

$t$=Control Parameter called as temperature in analogy with the actual annealing process.

Equation (1) shows that the probability of acceptance is a function of both the change in the cost function and the temperature. Initially temperature is set to a high value, which allows reasonable acceptance of worse moves. As the temperature decreases, the probability of accepting a worse move decreases and when it becomes zero, SA becomes a simple hill climbing technique. The algorithm has been given in Table 2.4.

There are two types of decisions in SA: the generic decisions; and the problem specific decisions [157]. The generic decisions mainly include the cooling schedule, which comprises the initial temperature, final temperature, change in temperature and the number of moves considered at each temperature. The initial temperature is set high enough to allow fairly free movement to any neighbourhood state. The change in temperature is critical to the success of algorithm. It must be changed systematically in order to avoid becoming trapped in a local optimum.

The problem specific decisions include the cost function, definition of the neighbourhood and the solution space. The cost function (also called the objective or fitness function) is used to measure the quality of a solution. It should be defined in such a way to guide the search process. Neighbourhood structure defines how the search moves from one solution to another in the solution space, which itself affects the number of iterations required to reach an optimal solution. Both categories of decisions have been further discussed in Chapter 3.

Select an initial solution $S_0$;
Select an initial temperature $t_0 > 0$;
Select a temperature reduction function $\alpha$;
Repeat

Repeat

Generate a move $S' \in N_{S_0}$;

Where $N_{S_0}$ defines the neighbourhood of $S_0$

$\delta = f_{(S')} - f_{(S_0)}$;

If $\delta < 0$

Then $S_0 = S'$;

Else

Generate random $x$ uniformly in the range (0, 1);

If $x < e^{-\delta/t}$ then $S_0 = S'$;

Until $innerLpCount = maxInnerLpNo$ or $f_{S_0}$satisfies the requirement;

Set $t = \alpha_t$;

Until $outerLpCount = maxOuterLpNo$ or $nonAcceptCount = maxNonAcceptNo$ or $f_{(S_0)}$ satisfies the requirement.

$S_0$ is the desired solution if $f_{(S_0)}$ satisfies the requirement.

Table 2.4: Standard SA algorithm

### 2.6.3 Tabu Search

Tabu Search (TS) is also a neighbourhood search technique first proposed by Glover in 1986 [65]. However, unlike SA where a randomized search is used, the searching process is more controlled in TS.

The main feature of the TS is its memory, where it keeps track of the searching process by recording some of the moves (or their attributes) that have already been investigated. This helps the search to move forward with out revisiting a move. Such moves that are not allowed to be revisited are called tabu and are kept usually in a list. The search process is iterative. The elements in the neighbourhood of the current state are evaluated and a decision is made as to where the search should move. The tabu list and the current state are updated.

To prevent the tabu list growing intractably, various memory strategies are used [67]. Recency functions are used which only keep recent moves in the list and discard the old ones, giving rise to what are usually referred to as short-term memory strategies. Similarly, there are a long term memory strategies. In *frequency-based* memory, a long-term memory strategy, the frequency with which a move occurs is stored and is used to penalize frequently visited moves.

There is an important exception to the tabu list. A move may be in a tabu list but it may still be selected. It may give rise to a better solution than any considered in the search so far. This is referred to as the aspiration criterion [45].

Other important concepts related to tabu search are intensification and diversification of the search. These are used to fine-tune the tabu search and are achieved by modifying the objective function. During intensification, the neighbourhood of a solution is searched more thoroughly. This is based on the belief that it may contain a better solution. This is achieved by penalizing the solutions which are far from the current solution. On the other hand to avoid large areas of search space remaining completely unexplored, diversification is also used.

Tabu search heuristics has been successfully applied to many optimisation problems. Current applications of TS span the realms of resource planning, telecommunications, VLSI design, financial analysis, scheduling, space planning,

energy distribution, molecular engineering, logistics, pattern classification, flexible manufacturing, waste management, mineral exploration, biomedical analysis, environmental conservation and scores of others [66].

### 2.6.4 Population Based Algorithms

Unlike previous methods where search centres around one candidate solution at a time, in population-based techniques a number of candidate solutions are kept. Most of these algorithms have been inspired by biological phenomena. Most of the terminology has been borrowed from there. Evolutionary Algorithms (EAs) form one group of such algorithms. The most common EAs are Genetic Algorithms (GAs) and Evolution Strategies (ESs).

#### Genetic Algorithms

Genetic Algorithms (GAs) were proposed in 1960s and 1970s by Holland [86] and his associates. These, as evident from the name, are bio-inspired search techniques, which mimic the principle of natural selection to find near optimal solutions to many computing problems.

In the natural selection processes, a species evolves by three main processes, i.e., selection, crossover and mutation over a long period of time. These processes are 'mimicked' in GAs. The basic GA algorithm is given in Table 2.5.

The algorithm begins by generating an initial population of solutions usually called chromosomes. The chromosome or the phenotype is usually represented using some encoding scheme, which is then called the genotype. Commonly used encodings are BCD, Gray encoding and two complement's etc. Much modern applications use a natural encoding scheme, i.e., the one closest to the phenotype.

The population is often initialised randomly. However, sometimes the population may also be seeded with some good solutions to facilitate the search process. After initializing population, each chromosome's fitness is measured with respect to some objective function that can be derived specific to the problem in context. Fitness values usually play important role in parent selection, where many methods can be used considering fitness in one way or another. The most popular

| |
|---|
| 1) Initialise a population of individuals (chromosomes)<br>Repeat<br>2) Calculate fitness of each individual in the population (relative to some objective function).<br><br>3) Select prospective parent (using selection methods).<br><br>4) Create new individual by mating parents (using crossover)<br><br>5) Mutate some of the individuals to introduce diversity.<br><br>6) Evaluate the new members and insert them into the population.<br>7) Until some termination condition is reached.<br><br>8) Return the best individual as the solution. |

Table 2.5: Standard GA algorithm

and well-studied methods are roulette wheel, tournament, and rank selection etc. [68].

In roulette wheel the parents are selected in proportion to their fitness values. However, the problem with this method is that it is biased towards parents having high fitness values, thus such parents will be selected in every generation. To get around this problem tournament selection method has been proposed. In this method, potential parents are selected randomly and then a tournament is held to select a parent for crossover. A variation of tournament selection is rank selection in which the population is first ranked and then every chromosome receives its fitness from this ranking. The parents are then selected on ranking, with a bias towards the higher ranks. This method usually leads to slower convergence.

Once parents are selected new individuals are produced by using the crossover or recombination and mutation operations. One point crossover operator was originally suggested, where a single point is selected and the portions of the parents to the right of this point are swapped. However, researchers have found other techniques perform better, e.g. multi-point crossover, uniform crossover operator, order-based crossover, partially matched crossover and cycled crossover. Parents may also be copied to the new population with out applying cross over operation

(Elitism). The reader is referred to [133, 153] for detail of these approaches.

Crossover may not necessarily lead to fitter offspring. Furthermore, crossover and selection alone may not be able to explore the whole solution space. To avoid this problem, the mutation operation is used. The operation usually consists of flipping a bit (or allele value generally) in the encoded individual with some small probability.

After these operations the new individuals are inserted into the next generation.

Other issues that are important in GA are the parameters, which include population size, number of parents, number of offspring, rate of crossover and rate of mutation. In most applications these parameters are selected based on trial-and-error.

The search process is terminated after some termination condition is reached, which is normally based on reaching an acceptable solution, time or number of iterations performed, or number of generations produced.

GAs have been used in many fields and are the most extensively used evolutionary algorithm used to solve many combinatorial optimisation problems. Applications range from industrial optimisation and design, neural network design, management and finance, artificial life, communication networks, electronics, and many others. GAs have also been applied successfully to generate test data. Xanthakis et al. [191] is considered the first work in this respect.

#### Evolution strategies

Another commonly used group of EAs is evolution Strategies (ESs) [156]. Developed almost in the same time span (or maybe earlier) as GAs, the ESs are very similar to GAs and differ only in the selection and 'breeding'. In GAs the dominant breeding operation is crossover, whereas in ESs it is mutation.

The two common ESs are the $(\mu, \lambda)$ and $(\mu + \lambda)$. In $(\mu, \lambda)$ ES, initially $\lambda$ number of individuals are generated randomly, from which, based on their fitness, $\mu$ number of parents are selected. Each of the $\mu$ parents is then mutated to generate $\lambda/\mu$ children. From the new generation again $\mu$ fittest individuals are

chosen and the process is repeated until some termination criterion is reached.

The $(\mu + \lambda)$ version is similar to $(\mu, \lambda)$ but instead of replacing parents with children, in $(\mu + \lambda)$, $\mu$ parents are also added in the next generation. The $(\mu + \lambda)$ version, though more exploitative as the parents are competing with children, may lead to premature convergence because of highly fit parents [108].

In comparison with GAs for real valued parameters, McTavish and Restrepo [123] (citing [147, 129]) reported that ES implementations are faster and more accurate.

### 2.6.5 Artificial Immune Systems (AIS)

Artificial Immune Systems are also bio-inspired algorithms which mirror the principles and processes of the natural immune system. AIS can be defined as "adaptive systems, inspired by theoretical immunology and observed immune functions, principles and models, which are applied to problem solving" [44]. From the definition it is clear that AIS are inspired from immunology, but do not slavishly simulate the process. Digital analogies of immune processes have been used to solve many different problems in computing. In order to understand AIS techniques, the following lines give a brief and highly simplified overview of basic immune system from [44].

The immune system is a self-controlled mechanism, which defends the body against the harmful organisms. The system consists of many components such as skin and saliva along with two forms of immunity i.e., innate and adaptive. Both these types complement each other. The innate immunity is unchanging. It provides initial defence against pathogens and also initiates and controls the adaptive immunity. The adaptive immunity modifies itself on exposure to unseen pathogens in order to enhance the defence. And hence because of its acquiring nature, this kind of immunity is more popular in the computing community for various problem solving tasks.

There are many different ways in which an immune system fights against pathogens. One such way is the production of antibodies, a kind of protein produced by a B-cell, which is a kind of white blood cell (lymphocyte). Antibodies bind to

antigens which are proteins produced by pathogens. The strength of this bonding is called affinity. Invading pathogens are recognised based on the strength of this affinity, which triggers an adaptive response to kill pathogens.

There are another kind of white blood cells i.e., T-cells which also play an important role in the immune system. T-cells are of three kinds: helper T-cells, killer T-cells and suppressor T-cells. Helper T-cells play main role in the activation of B-cells. Killer T-cells fight against foreign invaders whereas suppressor T-cells inhibit other immune cells and help in preventing allergic reaction and autoimmune diseases. The immune system consists of processes that help it to learn and remember antigens. Two theories attempt to explain these processes: the clonal selection and the immune network theory.

According to the clonal selection theory, recognition of antigen by antibodies triggers reproduction of associated B-cells resulting in a large number of clones of such cells. This process is called clonal expansion. The clones can undergo mutation with high rates, termed as somatic hypermutation. This leads to the change in the shape of the antibodies, which may result in improving the affinity of some antibodies to the antigens. Along with this B-cells can also differentiate into long-lived B memory cells. These memory cells, upon a second encounter with pathogens, proliferate into B-cell capable of producing antibodies, which has strong affinity with respective antigens.

According to network theory, there is a network of molecules, which constantly interact with each other irrespective of the presence of antigens and the properties such as learning, and memory is a result of the interaction of these network molecules. Antigen's surface contains regions called epitopes that are recognised by a set of paratopes on antibodies. A binding can also occurs between antibodies by the same mechanism. In this case instead of epitopes, there are ideotopes. This epitope-idetope interaction creates network structure, which includes stimulation of B-cells by other B-cells, hence causing their proliferation. In some cases suppression can also occur, which has a regulatory effect and hence lead to the death of cells.

Based on the above-mentioned processes in the natural immune system many

| |
|---|
| Initialise population of antibodies<br>WHILE (not finished)<br>Present antigen<br>Calculate fitness (by matching antigens and antibodies)<br>Select<br>Apply clonal expansion that reproduce new clones [Mutation (no crossover)]<br>Replace (variable population)<br>END WHILE |

Table 2.6: An AIS algorithm

AIS models are proposed, which extract and utilizes features like recognition, learning, memory, and predator-prey response etc. The key feature of most AIS models is the antibody to antigen match. A basic AIS algorithm [31] is shown in Table 2.6.

AIS is an evolving field and there is no fixed algorithm available. However, virtually all models have a population of B-cells, which are usually referred to as antibodies, without making any distinctions between the two. The antibodies represent the solution space where as, the antigen represents a target or desired solution. The antibodies and the antigen are then matched and decisions are made for cloning and mutation based on the affinity (matching) between the two, which can be based on some fitness function. If matching is good, new antibodies are added. Note that mutation may not be applied and depends upon the problem in context. The bad solutions are removed from the population and the process continues to find good solution until a termination condition is reached.

AIS has been successfully applied to a number of areas to solve many problems, which include computer security, data mining, robotics, machine learning, scheduling and optimization. However, as compared to other comparable areas such as neural networks, evolutionary algorithms and fuzzy systems, AIS lacks a general framework that can be applied to solve varied problems. Castro and Timmis [44] proposed a layered frame work, however much work need to be done to make it a formalized and established frame work.

### 2.6.6 Ant Colony Optimisation

Ant Colony Optimization (ACO) approaches are bio-inspired algorithms based on the working of real ants. The main underlying idea is that of parallelizing search based on a dynamic memory structure incorporating information on the effectiveness of previously obtained results and in which the behaviour of each single agent is inspired by the behaviour of real ants [110].

The inspiration for ACO was an experiment on Argentine ants where a colony of ant was connected with a food source through two paths of different length. The paths were arranged in such a way that any of the paths could be taken by ants with equal probability [54]. It was found that after some time, most of the ants started using the shortest path. This experiment was repeated by increasing the length of the longer path which resulted in decreased number of aunts taking that path.

The selection of shortest path is in fact the result of probabilistic decisions taken by ants based on indirect communication. The mean of indirect communication is pheromone, a chemical substance, deposited by ants while they walk. Ants can also sense this substance and their probabilistic choices are made by the amount of pheromone they sense. In the above-mentioned experiment, initially there was no pheromone on paths. However, because of the different length, ants following the shorter path reached the food source earlier and on their way back they smell the pheromone deposited by them. So they choose the shorter path with higher probability than the longer one. New pheromone is deposited on the path, increasing its amount, and thus making it more attractive for other ants and as the process continues, more and more ants took the shorter path. Another important element of this mechanism is the pheromone evaporation with time that enables the system to forget the past wrong decisions. On longer paths, the amount of pheromone deposited evaporates, reducing its amount. Whereas on frequently visited paths, the evaporation is counterbalanced by new pheromone added by ants.

The above-mentioned natural process has been adapted to develop a number of

algorithms for solving discrete optimisation problems. An earlier example is that of Ants System (AS) and its variants. These algorithms were further extended and modified, resulting in a general framework in the form of ACO.

ACO can be applied to discrete optimisation problems that can be characterised in graphical format. The main elements of the ACO are artificial ants that walk on a connected graph $G = (C, L)$, where nodes $C$ are connected by arc $L$. The feasible solutions to the problem correspond to the paths on $G$. Ants collect information during the search process which is encoded in pheromone trails associated with arcs, thus providing long-term memory holding information about the search. Arcs or nodes can also have an associated heuristic value giving a priori information on the problem. Each individual ant also has memory which is used to build feasible solutions, evaluate the solution found and retrace the path backward. Based on the information from pheromone trail, heuristic value and memory, ants make probabilistic decisions and add new paths to the solutions. Once a solution is complete, an ant retraces back the same path it followed. During this process it deposits pheromone in proportion to the quality of the solution. This help to direct the search process of the future ants.

Two more procedures i.e., pheromone trail evaporation and daemon actions (an optional component) are also included in ACO [54]. Pheromone trail evaporation reduces the amount of pheromone and thus helps avoid too rapid convergence of the algorithm towards a sub-optimal solution. Daemon actions are used to incorporate procedures in to the ACO, which cannot be performed by single ants. Such inclusion can be local search procedures or depositing additional pheromone based on the quality of solutions.

ACO has been successfully applied to a large number of combinatorial optimisation problems. These problems can be divided into two categories. The first category includes NP-hard optimisation problems where ACO is applied with local search algorithms to fine-tune the ant's solution. This category includes classical problems like Travelling Salesman Problem (TSP) and Quadratic Assignment Problem (QAP) etc. The second category includes problems where the problem instance changes from solution to solution during algorithm run time. Routing in

telecommunication networks is an example of such problems.

Intuitively, the use of ACO for dynamically changing problems seems well motivated. After all, the ant strategy has evolved to cater for the changing food sources. (Once a food source is depleted, the problem has changed!)

## 2.7 Automated Test Data Generation

Given a test requirement, a suitable set of input variables or test cases need to be developed to test it. Test data generation is, therefore, one of the most important steps in software testing. This activity is usually conducted manually. However, manual test data generation is a hard, laborious and very time consuming activity and contributes greatly to the software development and maintenance cost. Automating this step in testing can greatly reduce effort and cost of testing. There has been much research activity in this field with considerable success. Fawster [61] reported that saving of up to 80 percent over manual effort have been achieved.

There are many techniques for automation which are applied throughout the software development life cycle. We may broadly classify these techniques as random, static, and dynamic [150]. All the techniques have their advantages and disadvantages. However, in the context of this dissertation only the dynamic approach– i.e., search based test data generation is discussed.

### 2.7.1 Search Based Test Data Generation

In search based test data generation (SBTDG) achieving a test requirement is modeled as a numerical function optimisation problem and some heuristic is used to solve it. The techniques typically rely on the provision of "guidance" to the search process via feedback from program executions. For example, suppose we seek test data to satisfy the condition $X <= 20$. We can associate with this predicate a cost that measures how close we are to satisfying it, e.g. $cost(X <= 20) = max(X - 20, 0)$. The value $X == 25$ clearly comes closer to satisfying the condition than does $X == 50$, and this is reflected in the lower cost value associated with the former. The problem can be seen as minimising the cost

function $cost(X <= 20)$ over the range of possible values of $X$.

SBTDG for functional testing generally employs a search/ optimisation technique with the aim of causing assertions at one or more points in the program to be satisfied. We may require each of a succession of branch predicates to be satisfied (or not) to achieve an identified execution path; we may require the program preconditions to be satisfied but the postconditions to be falsified (i.e. falsification testing — finding test data that breaks the specification) [175]; or else we may simply require a proposed invariant to be falsified (e.g. breaking some safety condition, or causing a function to be exercised outside its precondition) [176].

### Survey

Search based test data generation can be dated back to 1976 [130]. However, it has been in recent years that interest developed greatly in the application of search based techniques to tackle the problem of software test data generation. A comprehensive survey of the different techniques is provided by McMinn [117]. In this section, the latest work in the field will be reviewed. More emphasis is placed on comparative and novel applications of search based techniques to the problem of software input test data generation.

Wegener [186] proposed a data flow based fitness function, however, the work by Girgis [64] seems to be the first quantitative work using data flow coverage measure. The criterion selected for coverage is $all - uses$. A standard GA is used with binary string encoding. The algorithm works by selecting a list of def-use paths, which are then given to the GA as input to be covered. The fitness function of a chromosome is calculated as the number of def-use paths covered by variable to the total numbers of def-use paths. Roulette wheel and random selection methods were used for selection parameters. The technique was tested on 15 small FORTRAN programs and 100 percent coverage was achieved in most of the programs. However, no information is given about the nature of the programs and their complexity. Also the proposed approach assigns the same fitness to the individuals which cover equal number of paths. It is possible that one individual may be better than the other in term of the nature of the path covered by it.

Michael and McGraw [127] compares local descent, standard simulated annealing and two variants of genetic algorithms i.e., standard GA and differential GA for more complex programs than previously studied. Condition/decision coverage is used as an adequacy criterion. Three sets of case studies are presented with increasing complexity. Random testing, as the case usually is, did much better for simple programs. However, overall standard SA and standard GA give much better results for complex programs, outperforming other approaches.

In their work, Alba and Chicano [14] used Evolutionary Strategies (ES) for test data generation. The proposed system uses condition decision coverage as adequacy measure and a distance based fitness function [176, 117] is used to guide the search. Information about which conditions have been reached or covered is stored in a decision table. In the former case, an ES is invoked to find the test data that cover the required decision. The condition table is also populated, if ES finds data that cover other decisions while searching for the required decision. Experimental results show that the ES performed slightly better than the GA. However, requiring less parameters to tune [14], ES seems a more attractive approach to generate test data.

Li and Lam [106] proposed an approach for test case sequence generation to satisfy all-state coverage criterion for statecharts using ant colony optimisation (ACO). A Statechart is converted into a directed graph and then a group of 'ants' is dispatched to cooperatively explore the graph using the 'pheronome' level left by the previous ants at neighbouring vertices. The effectiveness of this approach is not clear from the paper. However, this seems to be the first work using ACO to generate test sequences from statecharts.

Liaskos and Roper [107] proposed AIS to generate input test data using control flow coverage measure. A binary encoded scheme is used to encode the executed path (receptors), test targets (antigens) and test cases (immune cells). The fitness function (affinity) is the binary distance between the executed and target path. This is equivalent to the approximation or approach level [186] in traditional test data generation. The well-known Triangle program is used as a case study. However, the proposed algorithm, lacking the branch distance measurement does

not seem to perform well.

Sagarna and Lozano [164] includes application of Estimation of Distribution Algorithms (EDA) to software test data generation for branch coverage. The approach level with branch distance strategy is used to reach a branch. Different variants of EDAs suitable for discrete optimisation problems are considered. Results show that EDAs using nontrivial probabilistic models performed better. Further comparison is made with existing work on GAs. The authors conclude that the EDAs required fewer fitness evaluations to find the required test data in most cases as compared to GAs. In other work, Sagarna and Lozano [165] compared EDAs and Scatter Search. Based on their results a hybrid EDA-SS strategy was proposed.

Diaz et al. [52, 43] proposed a tabu search algorithm for the automatic generation of test data to obtain branch coverage. Using the branch coverage criteria of a CFG and distance based fitness function, the short term memory is used to store the best solutions. A long term memory tabu list is used to prevent local optimal entrapments by storing such solutions and avoiding them in the subsequent iterations. An approach similar to Korel's chaining approach [100] is adapted for selecting a target to be covered. Four neighbourhood solutions, half of which are in close proximity while half are farther away, are considered for each input variable in each move. This allows exploration of a wider neighbourhood. The approach is evaluated with three case studies using random testing as the bench mark and different ranges for input variables. Tabu search outperformed random testing in terms of speed and coverage. The approach seems plausible. However, for large and complex programs the short term memory list may contain too many solutions and the algorithm may become inefficient.

Currently most search based test data generation assumes branch predicates with numeric data types. Alshraideh and Bottaci [15] proposed a cost function for string equality, ordering and regular expression matching. The cost function and corresponding search operators are mainly based on information retrieval and biological string matching. Based on a number of small case studies they suggest that exploiting the presence of string literals in a program significantly improves

the performance of search based test generation for string data types.

Most of the time, the domain of application of search based test data generation has been procedural programming. Tonella [173] for the first time applied genetic algorithms to generate test cases for the unit testing of object oriented program. Unlike unit testing of procedural programs, in object oriented programming, an object needs to be created and brought in to a specific state before method invocation. To bring the object into a specific state may require the creation of other object(s) and method call(s). Thus a test case must cater for object/s and state creation in addition to the input test data. Tonella's approach captures this by defining a chromosome structure with two portions: one encode a sequence of statements and another the input values to be passed as parameters to the methods. Special mutation operators are defined to carry out operations on chromosomes. The fitness of a chromosome with respect to a target is obtained from the proportion of the control and call dependence edges that are traversed during the execution of the test cases. A test case which cover a target is saved and hence at the end of the algorithm a set of test cases is obtained, each of which contribute to the final level of coverage.

Wappler and Lammermann [184] proposed a more generic approach for the unit testing of object-oriented programs. Production rules are given for the test program, a phenotype, which takes into account object attributes in constructors as well as methods. The test program is encoded as a basic type value structure, a genotype individual. Each genotype is considered as a sequence of statements. Each statement is further structured into target object, method, and parameters. The target object and method are encoded as integers, whereas the encoding of a parameter depends on its type. In order to avoid unused genes and to optimize the evolutionary search, a multilevel optimization is used. Care is taken for the inconvertible genotypes and if the current population does not contain any convertible genotype, a mechanism is described to generate enough convertible genotypes in the proceeding population.

Wappler and Wegener [185] used strongly typed genetic programming to generate test cases for object oriented programs. They proposed using method call

| | |
|---|---|
| `{`<br>`flag=(a>b); ____ (i)`<br>`.......`<br>`if (flag)  ____ (ii)`<br>`{do something...}`<br>`}` | `{`<br>`flag=false`<br><br>`if (a>b); ____(iii)`<br>`flag=true;`<br>`if (flag)  ____ (iv)`<br>`{do something...}`<br>`}` |
| (a)Flag problem [26] | (b)More general case of Flag Problem. |

Table 2.7: Flag Problem

dependence graph to model call dependencies in a Java class. An acyclic subgraph of this graph corresponds to a feasible method call sequence which is modelled as tree. Strongly typed GP is used to manipulate these trees for generation of test programs.

### 2.7.2 Limitations of Search Based Test Data Generation

Although search based techniques have been successfully applied to generate input test data, still there are many areas where these techniques have limitations. McMinn [117] has comprehensively discussed those limitations. In this section we survey only the latest work regarding these problems.

#### Flag Problem

A flag is a boolean variable which if present in branch predicates can lead to straight plateaus in fitness landscape, thus degrading the search to a mere random search. To cope with the flag problem, various approaches have been suggested. Bottaci [27] considered the flag problem where the predicate value is evaluated at one point in the program and is used in another. Consider the code segment in Table 2.7.2 similar to the example in [117].

The flag is assigned value at (i), but is used at (ii), where the cost function is evaluated. But it is not useful as the boolean variable give either false or

true value leading to a flat fitness plateau. To get around the problem, Bottaci proposed an instrumentation strategy which retains information at such point as (i) in above code segment and then provides the respective cost value at (ii), where it is required. However, this is a particular case of the flag problem. The approach fails when a more general or complex form of flag problem is encountered. For example, for the code segment in Table 2.7.2(b), the cost calculated at (iii) cannot be propagated to (iv).

Harman et al. [76] proposed a testability transformation approach to solve the problem. Their approach is discussed in detail in Chapter 5

Baresel and Sthamer [21] suggested using static analysis to handle the flag problem. Their approach involves identifying 'decisive' branches i.e., the branches which has an effect on the flag assignment during the program execution. They divided the branches in to 'include' and 'exclude' lists. The execution of the branches in the 'include' list results in the desire flag assignment and vice versa. Unlike traditional fitness functions, which calculate fitness at a branch using the branch predicates only, they devised a fitness function to take into account all the branches affecting the flag assignment.

#### State problem

Another problem similar to the flag problem is the state problem [120]. The state problem occurs when information is stored in static internal variables, which are not directly accessible to the search process. The values of such variables can be changed only by executing the corresponding assignment statements. Often an input sequence is required for execution before bringing the system into the required state. In the presence of such state variables, the traditional search based testing approach does not perform better than random testing [120].

To cater for the state problem McMinn and Holcombe [119] proposed an extended version of 'chaining approach' [62]. In the chaining approach, a sequence of statements is identified, which is executed before the target node. The extended chaining approach identifies the set of all variables that can effect the 'problem' node. Event Sequences are then generated based on assignment to these variables,

which are then executed using a special encoding scheme to conduct structural testing of programs. The approach was applied successfully to a few programs achieving much higher coverage compared with a previous approach [21]. However, as discussed by the author, the chaining tree can become intractable making it very difficult to find feasible sequences. Also the approach suffers in the presence of nested branches and compound conditions.

Zhan and Clark [197] proposed a technique called 'tracing and deducing', to tackle the state problem in MATLAB Simulink models. To satisfy a test requirement, a back-propagation is sought to derive a more refined predicate structure that provide better guidance to the search process. The approach, however, requires the length of input sequence to be known in advance.

## 2.8 Conclusion

In this chapter we have presented a background literature relevant to this thesis. An overview of the software testing and adequacy criteria was given. It was followed by a brief discussion of different optimisation techniques that have been used to solve the problem of input test data generation. A survey of latest work was then given in the last section. It can be concluded that search based test data generation is an active research area where still many problems need to be solved.

Chapter 3

# Structural Testing: Searching for Fine Grained Test Data

## 3.1 Introduction

In structural or white box testing, a program is tested in order to locate the possible errors in a program.

To conduct structural testing, input test data is required. It can be generated manually, which is the normal practice in most of the organisations ([166], citing [57]). Manual generation of test data, however, is a laborious and time consuming activity and adds considerably to the overall cost of the project. Automation of this process is highly desirable. Many techniques have been proposed to automate the test data generation process [150].

One of the techniques that has gained popularity in recent years is search based test data generation, where the test data generation problem is modelled as a numerical optimisation problem. Data is then sought to solve this problem. The goal is usually to satisfy a coverage criterion.

A coverage criterion is important as it may help to detect errors that may cause the possible failure of the system. An overview of coverage criteria has been given in Chapter 2. In the case of search based test data generation, the most widely used coverage criterion has been branch coverage. However, this is not the strongest criterion and does not fulfill requirements for industrial standard

[162]. In this chapter a technique has been proposed to extend the existing search based test data generation techniques to generate more refined input test data. We have selected multiple condition coverage and MC/DC as coverage criteria to assess our technique. Interest in generating test data for more stronger criteria, using search based techniques, is growing. One recent work, carried out almost in parallel to ours, is by Awedikian et al. [20]. They also considered MC/DC as the coverage criterion. They considered an approach similar to the one proposed by McMinn and Holcombe [119] for state based programs, incorporating elements of control dependencies along with data dependencies to generate event sequence chains. This provides more guidance to the search. The work is similar to ours. However, it is more focused on the branch reachability problem.

In the following sections we describe the proposed framework and the experiments on tuning the parameters of simulated annealing, our chosen search technique. A section on evaluation considers the results of experiments conducted using the proposed framework.

## 3.2 Framework Implementation for Refined Test Data Generation

This section describes how search based techniques have been applied to generate test data for multiple condition coverage and MC/DC. We propose a framework. Figure 3.1 describes the high level architecture of the framework. The framework has three main parts: the instrumentor; the fitness function calculator; and the optimisation rig.

The Instrumentor takes the class under test (CUT) as input and produces an instrumented version. Paths and branches, for which data to be generated, are identified. Based on the coverage criteria, a fitness function is selected and then using the selected optimisation technique, the desired test data is searched. Java is used to implement the framework underpinning the work in this thesis. However, the research questions addressed in this thesis are not concerned with object-oriented aspects of the language. Although extension of the concepts explored

to encompass object-oriented elements is not precluded, this is not the subject of our research. (We would hope that any such extensions would be facilitated by our choice of Java as the implementation language.) The research questions largely target traditional imperative language structures. Java is used primarily for implementation convenience.

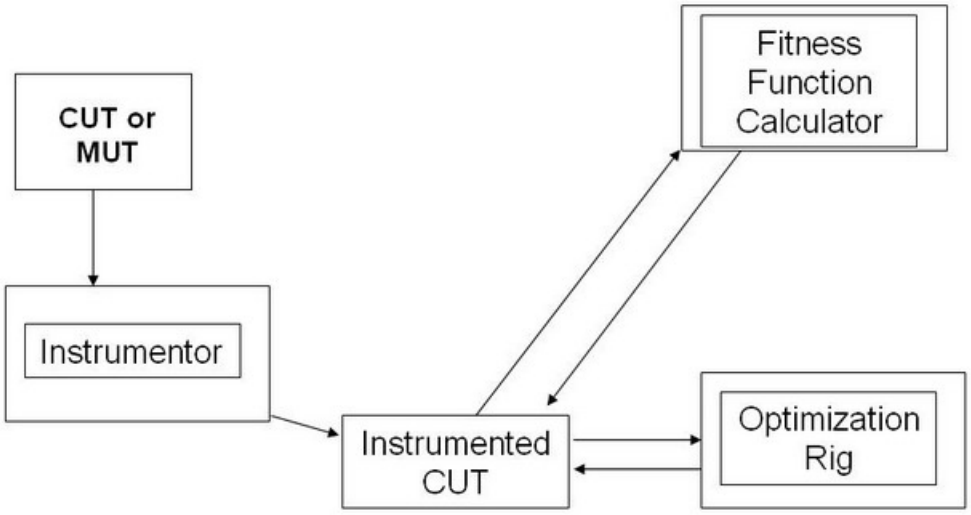


Figure 3.1: High Level Architecture of the Framework

### 3.2.1 Instrumentor

The Instrumentor is an important component of the framework. We used ANTLR [1] for Java parser generation using the Java grammar [132]. The Abstract Syntax Tree (AST) is generated, walked and instrumented at desired locations in the code. We used the Java emitter by Trip [177], which walks the modified AST to generate the instrumented code. The instrumentation scheme is designed to capture the information required for assessing how well the current test data achieves an identified purpose (i.e., the input to the fitness function), yet preserving the semantics of the program. The following lines explain the instrumentation scheme. All conditions are assumed to be in disjunctive normal form (DNF) [1].

If during parsing a control structure of the form

[1]It is always possible to reduce a condition to such a form [137]

$c_{00} \wedge c_{01}..c_{0n} \vee c_{10} \wedge c_{11}..c_{1n} \vee ..c_{n0} \wedge c_{n1}..c_{nn}$

is encountered, the instrumentor annotates it with the respective decision number, conjunct number and clause number.

Here

$c_{ij}$ represents $jth$ conjunct of $ith$ clause of the form

$(expr)relop(expr)$.

$relop$ can be any of the relational operators: $\leq, \geq, <, >, ==, != $

For example consider the following *if-branch*, the condition portion of which consists of two clauses and each clause consists of two conjuncts

$$if((x_1 < 15 \wedge x_2 > 10) \vee (y_1 > z_1 \wedge z_1 < 35))$$

The instrumentor replaces it by the following structure.

$if(data.complex(dec\#,$
$data.basic(x_1, \text{“}<\text{”}, 15, dec\#, clause_0, conjunct_0) \wedge$
$data.basic(x_2, \text{“}>\text{”}, 10, dec\#, clause_0, conjunt_1) \vee$
$data.basic(y_1, \text{“}<\text{”}, z_1, dec\#, clause_1, conjunct_0) \wedge$
$data.basic(z_1, \text{“}<\text{”}, 35, dec\#, clause_1, conjunct_0)))$

The instrumented code can then be compiled by any standard Java compiler. $data.basic()$ and $data.complex()$ are method calls to the object of a Data class which contains data structures to gather information from all instrumented branches during execution of the program.

The advantage of this kind of instrumentation is that we can store the information about input decision variables as well as the decision itself separately. This helps us to manipulate it in a flexible way for our purposes i.e., we can incorporate different cost functions and hence coverage criteria.

One problem that usually occurs with traditional source code instrumentation is that of undesirable side effects, which can alter the program's intended beha-

viour during run time. The suggested instrumentation scheme avoids this problem. However, if the program contains side effects, the instrumentation scheme will not make it side effect free. Techniques, such as those proposed by Harman et al. [82, 80], can be applied to obtain a side effect free version. In this work we assume all programs are side effect free.

### 3.2.2 Cost Function

We have chosen the approach level with branch distance strategy to reach the desired goal [117]. In this strategy, the cost function is devised in such a way to consider the path leading to the target. Thus the cost function is given as;

$$f(x) = CurrentDecisionCost+$$
$$K * NumberOfRemainingBranches$$

Where $K$ is constant and can be given any value just greater than the highest branch cost of any of the branches. The advantage of this value is that, it helps the search to be guided for a path that leads to the target.

When the current branch is the one directly leading to the the target without any other branches in between then the $BranchCost$ is the summation of costs of individual conditions in the MC/DC test case under consideration. For all other branches, leading to this branch, only the cost for branch coverage is calculated. The cost function has further been explained in Section 3.3.1.

The framework is implemented with routines to calculate cost functions for many different criteria. Currently it provides cost calculations for:

- For any decision it reaches.
- For any conjunct in the decision.
- For a series of specific conjunct truth assignments for each decision and
- For any identified series of branches (a path).

During the execution of the instrumented program, the framework also stores the minimum and maximum value of cost for a condition. This is particularly helpful for getting information about loop predicates as well as the branches inside a loop. For example, consider the following code segment:

```
public void loopTest(int x){
    for (int i=0;i<10; i++){
      x++;
      if (x>50){

       target statement;

      }
      else{

         statements;

      }

  }
}
```

The target is executed only when the value of $x$ is greater than 40. When the value is less than or equal to 40, the minimum fitness value guides the search to select a closer value until the goal is reached. If a false outcome of the branch is to be taken, then any value of $x$ less than 50 will reach the goal. However, if a value of 49 is taken, after the loop is executed, the value of x will be 59. If we do cost function calculation after the loop is executed, we would not be given a solution until the value of $x$ less than 41 is chosen. However, by keeping the minimum and maximum values, the framework avoids this problem.

### 3.2.3 Optimisation Rig

The framework is implemented in a modular form to incorporate different optimisation techniques. For proof of concept purposes we have used simulated

annealing in this work.

### 3.2.4 Required Test Case Sequence Generation

For multiple condition coverage, all possible combinations in the truth table are generated. For example, if we have a decision containing two conditions as $C1 \wedge C2$ then the following sequence of required truth value combinations is generated.

$True\ True$

$True\ False$

$False\ True$

$False\ False$

In the case of MC/DC, a minimal sequence of required combinations is generated. Mathur [113] describes a procedure to generate the minimal set of required combinations automatically for MC/DC coverage. However, only compound decisions containing a single kind of conditional operator was considered. We modified the technique to generate test sequences for compound decisions containing both conjuncts and disjuncts. The procedure is described below.

Consider a decision $D$ consisting of n conditions $C_i$, where $i = 1, 2, 3, ..., n$. To generate a required combination of MC/DC adequate test, the following steps are followed.

- Construct a table $T$ of size $(n + 1) \times (n)$. $T[i][j]$ is a cell in $T$ at row $i$ and column $j$, where $i = 0, 1, 2, ....n$ and $j = 0, 1, 2, ....n - 1$.
- Populate the cells $T[0][n - 1]$ and $T[1][n - 1]$ with $True$ and $False$ respectively. This gives us the test case sequence for one condition.
- Now we fill the immediate cells surrounding the already filled cells. Starting from the right check the first operator. If the operator is $OR$, then populate the surrounding cells $T[0][n - 2], T[1][n - 2], T[2][n - 2], T[2][n - 1]$ with $False$,$False$,$True$ and $False$, respectively. If the operator is $AND$ then populate the cells with $True$,$True$,$False$ and $True$ respectively. This gives us the test case sequence for two conditions.

- repeat above step increasing one more level. If the operator is the same as in previous step, then we get the required combination for three conditions. In case of a different operator, we change the cell value $T[2][n-1]$ to an opposite value.

- Repeat the above steps.

## 3.3 Tuning Parameters

As stated in Chapter 2 the factors affecting SA can be divided in to two categories (i) Problem specific (ii) Generic.

The problem specific decisions include cost function, search space and neighbourhood. Generic decisions, on the other hand, include initial and final temperature, the cooling schedule and the rate of cooling.

In search based software testing we found very little work regarding these factors. The only work that came to our knowledge is by Zhan [195] and a limited work by Tracey [176]. We conducted experiments to determine suitable values of simulated annealing parameters.

We chose three problems for this purpose as shown in Table 3.1. The respective programs have been shown in Appendix A. *NestedBr* is a custom written program with a constrained domain. It consists of three input variables and a branch nesting of two levels. The target is to generate test data for the inner most branch for the coverage of MC/DC test case *("True True True").*

*Triangle* is a benchmark program in software testing. Its description has been given in section 3.4. The target is to generate test data for equilateral triangle.

*Quad* has been taken from Zhan PhD thesis [195] where it was also used for SA parameters tuning. It has been written in code form. It consists of three input variables with a constrained domain. The objective is to generate test data that cover the false branch

$if((x - 310000)\text{x}(x - 310000) > 0.5)$

For all the three problems we select different ranges and data types ('int' and 'double') of input variables in order to pose different level of difficulty to the search

process. For double data type we chose an error limit $\delta$ of 0.0005.

Table 3.1: Problems For Parameters Settings

| Program | No of Vars | MC/DC Test Case | Range | Data Type | Search Space |
|---|---|---|---|---|---|
| NestedBr | 3 | T T T | $0 \sim 1500$ | int | $3.375\text{x}10^{09}$ |
| | | | $0 \sim 30000$ | int | $2.70\text{x}10^{13}$ |
| | | | $0 \sim 1500$ | double | $2.704\text{x}10^{19}$ |
| Triangle | 3 | T T | $-1000 \sim 1500$ | int | $1.56\text{x}10^{10}$ |
| | | | $-30000 \sim 30000$ | int | $2.16\text{x}10^{14}$ |
| | | | $-1000 \sim 1500$ | double | $1.25\text{x}10^{20}$ |
| Quad | 3 | T | $-5000 \sim 5000$ | double | $2.0\text{x}10^{7}$ |
| | | | $-500000 \sim 500000$ | double | $2.0\text{x}10^{9}$ |

### 3.3.1 Objective (Fitness) Function

In Simulated Annealing, fitness function is important in order to guide the search to an optimal solution. It gives a measure of 'goodness' of a solution with some chosen benchmark. For control flow coverage analysis, usually distance based objective functions are used. The value, usually called 'cost' of a solution, gives a measure of how far it is away from satisfying a coverage or 'sub' coverage criterion. Therefore, we usually refer to the objective function as the cost function. The cost function that we chose for this work is similar to the one proposed by Tracey [174]. It has been shown in Table 3.2. This a modified form of the work proposed by Korel [100]. The fitness function is based on evaluating branch predicates. It gives a value of 0 if the branch predicate evaluate to the desired value and a positive value otherwise. The lower the value, the better the solution.

### 3.3.2 Search Space

The search space includes all the possible solutions that are available to the search process. Thus if a problem contains input variables each with a certain range of values, then the search space is made up of all the combinations of these variables' values. In search based software testing the search space is usually driven by the requirements, therefore, is not be a factor to be tuned. However, it does effect other parameters such as neighbourhood proportion and hence can greatly effect

Table 3.2: Cost Function

| Predicate | Value of Cost Function F |
|---|---|
| Boolean | if TRUE then 0, else K |
| $E_1 < E_2$ | if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + K$ |
| $E_1 \leq E_2$ | if $E_1 - E_2 \leq 0$ then 0, else $E_1 - E_2$ |
| $E_1 > E_2$ | if $E_2 - E_1 \leq 0$ then 0, else $E_2 - E_1 + K$ |
| $E_1 \geq E_2$ | if $E_1 - E_2 \leq 0$ then 0, else $E_2 - E_1$ |
| $E_1 = E_2$ | if $abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2) + K$ |
| $E_1 \neq E_2$ | if $abs(E_1 - E_2) \neq 0$ then 0, else $K$ |
| $E_1 \vee E_2$ ($E_1$ unsatisfied, $E_2$ unsatisfied | $min(cost(E_1), cost(E_2))$ |
| $E_1 \vee E_2$ ($E_1$ unsatisfied, $E_2$ satisfied | 0 |
| $E_1 \vee E_2$ ($E_1$ satisfied, $E_2$ unsatisfied | 0 |
| $E_1 \vee E_2$ ($E_1$ satisfied, $E_2$ satisfied | 0 |
| $E_1 \wedge E_2$ ($E_1$ unsatisfied, $E_2$ unsatisfied | cost($E_1$)+cost($E_2$) |
| $E_1 \wedge E_2$ ($E_1$ unsatisfied, $E_2$ satisfied | cost($E_1$) |
| $E_1 \wedge E_2$ ($E_1$ satisfied, $E_2$ unsatisfied | cost($E_2$) |
| $E_1 \wedge E_2$ ($E_1$ satisfied, $E_2$ satisfied | 0 |

search efficiency. Therefore due consideration is given to search space while tuning other parameters. In our test problems for tuning parameters, we have chosen different search spaces from small to very large.

### 3.3.3 Neighbourhood

In search based test data generation, the neighbourhood is the maximum range, called the *'Move Strategy'* [195], within which a randomly chosen input variable is incremented or decremented with a randomly chosen value in order to make a move. This is usually taken as some fraction of the range of the input variables' domain. An appropriate value for move step-size is important. The search process will take too long if a step sizes is small. On the other hand, a large step-size may cause the search process merely sampling from a large portion of search space may not be able to amply explore the area where the solution lies [38].

There are usually two approaches applied for choosing a step-size: the 'fixed-strategy' and the 'dynamic or variable-strategy' [193, 195]. In the fixed-strategy, the step-size remains constant through out the search process. In case of the variable-strategy, the step-size changes as the search proceeds. The concept is based on the idea that during the initial temperature, when the probability of accepting 'bad' moves is very high, exploration of the search space is important and hence a large step-size is required. On the other hand, at low temperature some areas of the search space need to be exploited and hence a smaller move step-size is more suitable [193].

In the domain of software testing, only Zhan [195] has conducted a study with both strategies. Zhan experimented with step-sizes of 2% and 0.2% of the input variables' domain for fixed-strategy. Experiments for variable-strategy conducted with initial step-sizes of 20% and 2% and were allowed to decrease with the same rate as the cooling temperature rate with each iteration of the outer loop. In most cases, the fixed-strategy with larger step-size gave better results. The variable strategy performed very poorly. However, this has been because of the fast decrease in the move step-size. For example, for an initial move step-size of 2%, temperature decrease rate of 0.8 and a cooling schedule of 100×300

(innerLoops×OuterLoops), the step-size becomes about 0.00002% of the search space only after fifty outer loop iterations. This makes it almost impossible for the algorithm to find a solution if a solution is not found during the intitial iterations. Therefore, the author suggested using longer inner loop iterations and a smaller move step-size decrease.

Tracey [176] also performed experiments with the fixed-strategy. His study does not show much effect of this parameter on the performance of the algorithm. However, the performance measure concerned coverage achievements only and no studies were conducted to compare the success rate and number of fitness function executions (i.e., the efficiency).

### 3.3.4 Temperature

Two important factors related to temperature are initial temperature and final temperature at which the SA terminates.

The initial temperature is usually high enough, so that the acceptance probability is close to one. So almost every move is accepted. However, a very high initial temperature may cause a long computation time or bad performance [151].

Kirkpatrick et al. [98] suggested an initial temperature which allows a fraction of all the 'bad' moves to be accepted according to an initial acceptance ratio $\chi_o$. $\chi_o$ is the ratio of moves accepted to the total number of moves made. The procedure proposed by Kirkpatrick et al. is to chose an initial high temperature randomly and perform a number of random moves. The ratio of accepted moves is compared with $\chi_o$. If it is less than $\chi_o$, then the temperature is multiplied by 2. The process is repeated until the ratio exceeds $\chi_o$. This is generally referred to as 'temperature doubling'.

We also find many other schemes in SA literature for initial temperature such as those proposed by Johnson et al. [92], Park and Kim [151], Ben-Ameur [23]. In this thesis, we have used the above mentioned approach by Kirkpatrick et al.

As the SA algorithm progresses, the temperature cools. At low temperature, 'bad' moves are accepted with low probability. The process continues until some termination criterion is reached. Many termination criteria have been proposed.

A preferred method is to stop SA when a predetermine maximum count of the inner loop iterations is reached. Johnson et al. [92] as cited in Park and Kim [151] proposed a method, which increments a counter by one, when no 'bad' move is accepted during one outer loop iteration. If a move is accepted, the counter is reset to zero. SA is stopped, when the counter exceeds a predetermined value. In our work we have implemented both approaches.

In search based testing literature, Zhan [195] proposed an initial temperature of 0.7, Mansour and Salame [111] proposed a value of 0.90 whereas Tracey [176] chose a value of 0.95. However, it is not clear why these values were selected.

For final temperature, Zhan and Tracey considered a predetermined iterations of inner loop as the stopping criterion. Mansour and Salame [111] chose a final temperature of $2^{-30}$ as the stopping criteria.

### 3.3.5 Cooling Schedule and rate

The cooling schedule constitutes the number of trials, at which the temperature remains constant, before it is cooled down to a lower value at a certain cooling rate. This is also called as Markov chain length [38] or epoch length [151].

The Markov chain length can be set proportional to the size of the problem or the size of the neighbourhood [151]. However, in software testing, the neighbourhood size can be infinite as in case of a 'double' value solution [195]. Therefore, a few trial sizes are experimented with before selecting the final size.

The cooling rate $\alpha$ is also an important parameter. A high cooling rate will cause SA to converge too quickly without enough exploration of the search space, thus causing it to stuck in a local minimum. On the other hand a low cooling rate can be computationally expensive as it will take too long to converge to a solution. It is important to choose a cooling rate, which allow enough exploration and exploitation of the search space.

Many schedules have been proposed. The most common one involves multiplying temperature $T_i$ by $\alpha$ at each Markov chain length to get the following temperature $T_i$ i.e., $T_i = \alpha T$. In SA literature usually a value 0.8 to 0.99 is suggested for $\alpha$ [187, 157]. We also carried out experiments in this range.

### 3.3.6 Experimentation Results for Parameters Setting

The results of the experiments have been shown in the following lines with the a conclusion at the end of this section.

#### Initial Threshold

For initial threshold, following experimental set up was used.

- initial Temperature=100;
- Move Strategy: Fixed with a parameter of 0.005;
- Cooling rate : 0.8
- Inner loop iterations($I_L$): 100
- Outer loop Iterations($O_L$): 1500
- Termination : Either data is found or no move is accepted for $150 \times I_L$ or $I_L \times O_L$ is reached.

Table 3.3 summarises the results. The values in cells indicate the cost of search in terms number of fitness function evaluations and the corresponding success rate, which has been given in brackets.

In case of *NestedBr* initial acceptance ratio had little effect. An initial Threshold of 0.6 slightly perform better than other values. For *Triangle* and *Quad*, the search performed better mostly at a value of 0.6. In all the experiments the search was usually more expensive at higher threshold values.

#### Move Strategy

For *fixed* strategy, we experimented with four step-sizes i.e., 0.5%,1%, 2% and 5% of input variable range. Following experimental set up was used.

- Initial Temperature:100
- Initial threshold: 0.6

Table 3.3: Initial Threshold at a cooling rate of 0.8 and variance 0.005

| Program | Range | Initial Threshold | | | | |
|---|---|---|---|---|---|---|
| | | 0.4 | 0.6 | 0.8 | 0.9 | 1.0 |
| NestedBr | I | 4955(100%) | 5336(100%) | 5182(100%) | 5104(100%) | 5436(100%) |
| | II | 4726(100%) | 3975(100%) | 4065(100%) | 4349(100%) | 4687(100%) |
| | III | 4397(100%) | 4320 (100%) | 5191(100%) | 4851(100%) | 5504(100%) |
| Triangle | I | 4026(100%) | 4579(100%) | 3836(100%) | 3875(100%) | 4573(100%) |
| | III | 30628(60%) | 25377(73.33%) | 30152(73.33%) | 29193(46.67%) | 31195(46.67%) |
| Quad | I | 1302(100%) | 889(100%) | 1844(100%) | 2087(100%) | 1930(100%) |
| | II | 8763(100%) | 5416(100%) | 6390(100%) | 10188(100%) | 7428(100%) |

- Move Strategy: Fixed with a parameter of 0.005, 0.01, 0.02 and 0.05
- Cooling rate : 0.8, 0.9 and 0.95
- $(I_L)$x$(O_L)$: 250×600, 100×1500
- Termination : Either data is found or no move is accepted for $60\times I_L$ and $150\times I_L$ respectively for 250×600 and 100×1500 or maximum limit of iterations is reached.

Tables 3.4, 3.5, and 3.6 show the results. In case of problems *NestedBr* and *Triangle*, for short ranges of input variable, a step-size of 0.05 gave much better results. Whereas 0.005 gave better results for large input ranges. For *Quadratic*, for shorter input range, a step-size of 0.02 gave much better results, whereas step-size of 0.005 was dominant in case of the larger input range. Thus in case of fixed strategy, for large input spaces, it is suggested to use a small step-size and vice versa.

For variable neighbourhood strategy, three initial step sizes were used i.e., 2%, 10% and 20% of input variable range. Following experimental set up was used.

- Initial Temperature:100
- Initial threshold: 0.6
- Move Strategy: Variable with a parameter of 0.02, 0.1 and 0.2
- Cooling rate : 0.8
- $(I_L)$x$(O_L)$: 250×600, 100×1500
- step-size decrease rate: 0.95 at $10I_L$ of 100×1500 schedule for small search spaces and $25I_L$ for large search spaces.
- Termination : Either data is found or no move is accepted for $60\times I_L$ and $150\times I_L$ for 250×600 and 100×1500 schedule respectively or maximum limit of iterations is reached.

In addition to the usual SA parameters, another parameter i.e., *step-size decrease rate* is also required to be set up. With limited experiments, we chose such values in order to allow sufficient iterations at a reasonable step sizes. Table 3.7 summarises the results. It can be seen that the search performed better at shorter step-size of 0.02 for both larger and smaller search spaces.

Comparing fixed and variable strategies, we can see from Tables 3.4, 3.5, 3.6, and 3.7 that there is no clear winner. Therefore, we selected fixed strategy with shorter Markov chain length for further experiments.

### Cooling Rate

For cooling rate we performed experiments for fixed strategy only. The results are concluded from the same experimental set up as for fixed strategy. As can be seen from Tables 3.4, 3.5, and 3.6, a cooling rate of 0.8 performed much better and therefore chosen in our experiments.

### Markov Chain Length

For Markov chain length, we chose two different schedules *short* and *long*. For short schedule, a length of 100×1500 and for long, a length of 250×600 was chosen. As evident from Tables 3.4, 3.5, and 3.6, the shorter length Markov chain performed best most of the time and therefore is chosen for further experiments in this thesis.

Table 3.4: Experiment Results for a fixed neighbourhood strategy at a cooling rate of 0.8

| Program | Variance | Range | 250×600 | | 100×1500 | |
|---|---|---|---|---|---|---|
| | | | Executions | Success | Executions | Success |
| NestedBr | 0.01 | I | 4719 | 100% | 2972 | 100% |
| | | II | 4440 | 100% | 2812 | 100% |
| | | III | 4496 | 100% | 3157 | 100% |
| | 0.02 | I | 3400 | 100% | 1915 | 100% |
| | | II | 6853 | 100% | 3538 | 100% |
| | | III | 3143 | 100% | 1802 | 100% |
| | 0.05 | I | 2716 | 100% | 1513 | 100% |
| | | II | 10782 | 100% | 6005 | 100% |
| | | III | 2470 | 100% | 1206 | 100% |
| | 0.005 | I | 7408 | 100% | 4507 | 100% |
| | | II | 4782 | 100% | 3953 | 100% |
| | | III | 8095 | 100% | 4752 | 100% |
| Triangle | 0.01 | I | 3946 | 100% | 2020 | 100% |
| | | II | 7539 | 86.67% | 4326 | 100% |
| | | III | 32326 | 40% | 32055 | 50% |
| | 0.02 | I | 3242 | 100%% | 1745 | 100% |
| | | II | 8919 | 93.33% | 4872 | 100% |
| | | III | 48317 | 30% | 29584 | 30% |
| | 0.05 | I | 3147 | 100% | 1719 | 100% |
| | | II | 12249 | 100% | 8560 | 100% |
| | | III | 43150 | 0% | 38110 | 0% |
| | 0.005 | I | 6973 | 100% | 3819 | 100% |
| | | II | 4862 | 100% | 3496 | 100% |
| | | III | 33510 | 53.33% | 33834 | 53.33% |
| Quad | 0.01 | I | 221 | 100% | 252 | 100% |
| | | II | 6786 | 100% | 7558 | 100% |
| | 0.02 | I | 216 | 100% | 140 | 100% |
| | | II | 7452 | 100% | 10928 | 96.67% |
| | 0.05 | I | 261 | 100% | 220 | 100% |
| | | II | 22941 | 73.33% | 23878 | 50% |
| | 0.005 | I | 386 | 100% | 460 | 100% |
| | | II | 2966 | 100% | 4096 | 100% |

Table 3.5: Experiment Results for a fixed neighbourhood strategy at a cooling rate of 0.9

| Program | Variance | Range | 250×600 | | 100×1500 | |
|---|---|---|---|---|---|---|
| | | | Executions | Success | Executions | Success |
| NestedBr | 0.01 | I | 7630 | 100% | 4505 | 100% |
| | | II | 9724 | 100% | 3886 | 100% |
| | | III | 7465 | 100% | 4098 | 100% |
| | 0.02 | I | 5740 | 100% | 2911 | 100% |
| | | II | 11581 | 100% | 6377 | 100% |
| | | III | 4733 | 100% | 2450 | 100% |
| | 0.05 | I | 4423 | 100% | 2219 | 100% |
| | | II | 18012 | 100% | 10380 | 100% |
| | | III | 3943 | 100% | 1814 | 100% |
| | 0.005 | I | 11002 | 100% | 6732 | 100% |
| | | II | 7406 | 100% | 6775 | 100% |
| | | III | 10871 | 100% | 6234 | 100% |
| Triangle | 0.01 | I | 5855 | 100% | 3106 | 100% |
| | | II | 9027 | 100% | 5221 | 100% |
| | | III | 60121 | 30% | 48134 | 13.33% |
| | 0.02 | I | 4947 | 100% | 2205 | 100% |
| | | II | 12455 | 100% | 9193 | 100% |
| | | III | 49425 | 13.33% | 43742 | 30% |
| | 0.05 | I | 5118 | 100% | 2704 | 100% |
| | | II | 19964 | 93.33% | 13010 | 100% |
| | | III | 53471 | 13.33% | 41627 | 20% |
| | 0.005 | I | 10045 | 100% | 6258 | 100% |
| | | II | 8896 | 100% | 4081 | 100% |
| | | III | 38571 | 80% | 36245 | 53.33% |
| Quad | 0.01 | I | 181 | 100% | 228 | 100% |
| | | II | 4763 | 100% | 5768 | 100% |
| | 0.02 | I | 193 | 100% | 218 | 100% |
| | | II | 18107 | 100% | 9734 | 100% |
| | 0.05 | I | 276 | 100% | 395 | 100% |
| | | II | 26621 | 86.67% | 19620 | 96.667% |
| | 0.005 | I | 387 | 100% | 717 | 100% |
| | | II | 3602 | 100% | 3756 | 100% |

Table 3.6: Experiment Results for a fixed neighbourhood strategy and cooling rate of 0.95

| Program | Variance | Range | 250×600 | | 100×1500 | |
|---|---|---|---|---|---|---|
| | | | Executions | Success | Executions | Success |
| NestedBr | 0.01 | I | 10841 | 100% | 6335 | 100% |
| | | II | 12018 | 100% | 7153 | 100% |
| | | III | 10437 | 100% | 6328 | 100% |
| | 0.02 | I | 8923 | 100% | 4618 | 100% |
| | | II | 21329 | 100% | 10126 | 100% |
| | | III | 6669 | 100% | 4298 | 100% |
| | 0.05 | I | 7440 | 100% | 4626 | 100% |
| | | II | 28721 | 100% | 15281 | 100% |
| | | III | 5549 | 100% | 3292 | 100% |
| | 0.005 | I | 17730 | 100% | 10451 | 100% |
| | | II | 12749 | 100% | 5958 | 100% |
| | | III | 16717 | 100% | 9547 | 100% |
| Triangle | 0.01 | I | 9519 | 100% | 4596 | 100% |
| | | II | 14292 | 93.33% | 7654 | 100% |
| | | III | 69548 | 40% | 45049 | 20% |
| | 0.02 | I | 8390 | 100% | 4385 | 100% |
| | | II | 20631 | 93.33% | 10263 | 93.33% |
| | | III | 75902 | 13.33% | 51176 | 13.33% |
| | 0.05 | I | 5846 | 100% | 5304 | 100% |
| | | II | 31402 | 100% | 14482 | 100% |
| | | III | 62356 | 30% | 44454 | 13.33% |
| | 0.005 | I | 22955 | 100% | 8019 | 100% |
| | | II | 12289 | 100% | 5539 | 100% |
| | | III | 58364 | 70% | 30111 | 80% |
| Quad | 0.01 | I | 216 | 100% | 212 | 100% |
| | | II | 4252 | 100% | 6703 | 100% |
| | 0.02 | I | 170 | 100% | 174 | 100% |
| | | II | 7324 | 100% | 15145 | 100% |
| | 0.05 | I | 255 | 100% | 264 | 100% |
| | | II | 29976 | 86.67% | 26287 | 80% |
| | 0.005 | I | 299 | 100% | 687 | 100% |
| | | II | 4327 | 100% | 3108 | 100% |

Table 3.7: Experiment Results for a Variable neighbourhood strategy

| Program | Initial Step-Size | Range | 250×600 | | 100×1500 | |
|---|---|---|---|---|---|---|
| | | | Executions | Success | Executions | Success |
| NestedBr | 0.1 | I | 3216 | 100% | 1528 | 100% |
| | | II | 13535 | 100% | 10320 | 100% |
| | | III | 2325 | 100% | 1264 | 100% |
| | 0.2 | I | 4893 | 100% | 2192 | 100% |
| | | II | 20508 | 100% | 15948 | 100% |
| | | III | 4379 | 100% | 1758 | 100% |
| | 0.02 | I | 3570 | 100% | 1951 | 100% |
| | | II | 6369 | 100% | 3603 | 100% |
| | | III | 3276 | 100% | 1882 | 100% |
| Triangle | 0.1 | I | 4712 | 100% | 2054 | 100% |
| | | II | 24025 | 100% | 17535 | 100% |
| | | III | 52054 | 40% | 51806 | 53.33 |
| | 0.2 | I | 7252 | 100% | 2884 | 100% |
| | | II | 28022 | 100% | 24186 | 100% |
| | | III | 71924 | 66.67% | 62737 | 46.67% |
| | 0.02 | I | 3065 | 100% | 1692 | 100% |
| | | II | 6992 | 100% | 7143 | 100% |
| | | III | 41827 | 73.33% | 56644 | 33.33% |
| Quadratic | 0.1 | I | 393 | 100% | 443 | 100% |
| | | II | 21315 | 100% | 43328 | 93.33% |
| | 0.2 | I | 937 | 100% | 1852 | 100% |
| | | II | 22762 | 93.33% | 27649 | 60% |
| | 0.02 | I | 198 | 100% | 156 | 100% |
| | | II | 5546 | 100% | 8863 | 100% |

## 3.4 Experimentation

This section describes experiments conducted to generate test data for MC/DC and multiple condition coverage. Test data is also generated using random testing. Both techniques are then compared and analysed. The aim of the experiments is to provide evidence that:

- Search based test data generation techniques can be extended to generate test data for MC/DC and multiple condition coverage.
- The proposed technique is a more efficient means than random testing for generating test data to achieve coverage of more complex structures.

### 3.4.1 Test Objects

We performed experiments with benchmark testing programs which are taken mostly from the current software testing literature. Following test objects are used for evaluating the proposed approach. The test objects have been given in Appendices A and B.

#### Triangle

Triangle is a benchmark program and has been used many times in many works related to software testing. It takes three input variables as sides of a triangle and decides whether these constitute a triangle and if so, the type of the triangle. A variant of this program was used that contains more complex branching structures. All the variables are of type integer and their range is from -1000 to 1500.

#### CalDate

CalDate is taken from May's PhD thesis [115]. This program converts the date given as day, month and year into a Julian date (not including the fractional time part). The Julian date is then calculated as the number of days from the 1st January, 4713 B.C. If the date to be converted is after 15th October 1582,

which is the date of introduction of the Gregorian calender, then the Julian date is adjusted. The Julian date is then returned.

#### Quadratic

Quadratic program determines the roots of a quadratic equation. It takes three integer input variables in the range -1000 and 1000 and determines whether the resulting equation is quadratic and if so, whether it has real or complex roots.

#### Complex

Complex is an artificial program with difficult branching structure. It consists of five input variables in the range -1000 to 1500. The purpose of the program is to check the MC/DC test data generation for difficult nested branches.

#### Expint

Exponential Integral Program, is based on *'expint'* routine from [180]. It consists of two input variables, First is of type integer and the second one is of type double in the range -1500 to 1500.

### 3.4.2 Experimental Setup

We used the following parameters for simulated annealing based on the experiments in the previous section.

Move strategy: fixed

Initial temperature threshold: 0.6

Geometric temperature decrease rate: 0.8

Number of iteration in inner loop: 100

Maximum number of iterations in outer loop: 1500

Stopping criteria: Either a solution is found or maximum number of iterations (100x1500) is reached or no move is made for 150x$I_L$.

### 3.4.3 Analysis

To evaluate the performance of the test data generation for MC/DC and multiple condition coverage, random testing is also carried out as a baseline testing approach.

Results of the experiments are presented in Table 3.9 and 3.8. All results are averaged over thirty runs of each program. Coverage is calculated by dividing the number of required combinations covered by test data to the total number of required feasible combinations. It is shown in columns 4 and 7, in both tables for random testing and SA. Columns 5 and 8 shows the success rate, which is calculated by dividing the number of time the data is found to the total number of runs. The average number of executions taken to search the required test data is shown in columns 6 and 9 respectively. In the following lines we discuss the results for each program.

For Triangle, SA did poorly for the first and second branch in both multiple condition and MC/DC coverage criteria. However, performed much better for third condition, which required test data generation for equilateral triangle.

CalDate was an easy program with a limited search space of input variables. Same required combination of test case sequences was required to satisfy both multiple condition and MC/DC coverage. Random testing and SA perform equally better.

For Quadratic program again random testing perform better than SA. Same required combination of test sequences for MCDC and condition coverage was also required here. Random testing was more successful in finding the data for the third condition, where SA stuck more frequently in local minima.

SA outperformed random testing in case of Complex program. Random testing especially had difficulty with finding test data for the last two decisions. SA was much more efficient in terms of cost function evaluation as well as success rate.

In case of Expint, condition 1 was most difficult. SA performed much better in this case as well.

Table 3.8: Test Data Generation for Multiple Condition Coverage

| Program | Decision No | No of Test Cases | Random | | | SA | | |
|---|---|---|---|---|---|---|---|---|
| | | | Coverage(%) | Success(%) | Exec | Coverage(%) | Success(%) | Exec |
| Triangle | 0 | 8 | 100 | 100 | 31 | 100 | 100 | 2264 |
| | 1 | 8 | 100 | 100 | 75045 | 100 | 100 | 77539 |
| | 2 | 4 | 75 | 75 | 48274 | 100 | 100 | 5064 |
| | 3 | 8 | 100 | 100 | 81154 | 100 | 100 | 77660 |
| CalDate | 0 | 2 | 100 | 100 | 22 | 100 | 100 | 7 |
| | 1 | 2 | 100 | 100 | 96 | 100 | 100 | 90 |
| | 2 | 2 | 100 | 100 | 45 | 100 | 100 | 115 |
| Quadratic | 0 | 4 | 100 | 100 | 1935 | 100 | 100 | 1924 |
| | 1 | 4 | 100 | 100 | 6 | 100 | 100 | 339 |
| | 2 | 4 | 100 | 60 | 71488 | 100 | 51.67 | 70863 |
| Complex | 0 | 8 | 100 | 100 | 119 | 100 | 100 | 7110 |
| | 1 | 32 | 91 | 82.5 | 66106 | 100 | 100 | 54554 |
| | 2 | 8 | 42 | 3.81 | 147779 | 100 | 100 | 28828 |
| | 3 | 4 | 0 | 0 | 150000 | 100 | 100 | 49522 |
| Expint | 0 | 32 | 50 | 50 | 113399 | 100 | 99.58 | 82857 |
| | 1 | 2 | 100 | 100 | 3799 | 100 | 100 | 2390 |
| | 2 | 2 | 50 | 50 | 75008 | 100 | 98.33 | 19375 |
| | 3 | 2 | 100 | 100 | 4331 | 100 | 100 | 2958 |
| | 4 | 2 | 50 | 50 | 78054 | 100 | 100 | 4045 |
| | 5 | 2 | 100 | 100 | 79158 | 100 | 100 | 2916 |
| | 6 | 2 | 100 | 35.33 | 77375 | 100 | 100 | 2861 |
| | 7 | 2 | 100 | 100 | 9709 | 100 | 100 | 3149 |
| | 8 | 2 | 0 | 0 | 150000 | 100 | 100 | 2887 |
| | 9 | 2 | 100 | 100 | 19 | 100 | 100 | 1324 |
| | 10 | 2 | 100 | 100 | 12 | 100 | 100 | 2615 |

Table 3.9: Test Data Generation for MC/DC

| Program | Decision No | No of Test Cases | Random | | | SA | | |
|---|---|---|---|---|---|---|---|---|
| | | | Coverage(%) | Success(%) | Exec | Coverage(%) | Success(%) | Exec |
| Triangle | 0 | 4 | 100 | 100 | 35 | 100 | 100 | 2736 |
| | 1 | 4 | 100 | 100 | 88 | 100 | 100 | 5049 |
| | 2 | 3 | 100 | 67.67 | 60498 | 100 | 100 | 3767 |
| | 3 | 4 | 100 | 100 | 12583 | 100 | 100 | 4436 |
| CalDate | 0 | 2 | 100 | 100 | 14 | 100 | 100 | 9 |
| | 1 | 2 | 100 | 100 | 76 | 100 | 100 | 83 |
| | 2 | 2 | 100 | 100 | 57 | 100 | 100 | 132 |
| Quadratic | 0 | 2 | 100 | 100 | 2363 | 100 | 100 | 865 |
| | 1 | 2 | 100 | 100 | 5 | 100 | 100 | 499 |
| | 2 | 2 | 100 | 63.33 | 69022 | 100 | 53.33 | 65301 |
| Complex | 0 | 4 | 100 | 100 | 235 | 100 | 100 | 5663 |
| | 1 | 6 | 83.33 | 65 | 69967 | 100 | 87.77 | 24525 |
| | 2 | 4 | 0 | 0 | 150000 | 100 | 100 | 8639 |
| | 3 | 3 | 0 | 0 | 150000 | 100 | 100 | 12188 |
| Expint | 0 | 6 | 50 | 50 | 77732 | 100 | 100 | 13224 |
| | 1 | 2 | 100 | 100 | 3540 | 100 | 100 | 2530 |
| | 2 | 2 | 50 | 50 | 75005 | 100 | 100 | 18868 |
| | 3 | 2 | 100 | 100 | 3473 | 100 | 100 | 2576 |
| | 4 | 2 | 100 | 100 | 78786 | 100 | 100 | 3047 |
| | 5 | 2 | 100 | 100 | 79158 | 100 | 100 | 2924 |
| | 6 | 2 | 100 | 51.667 | 78446 | 100 | 100 | 3196 |
| | 7 | 2 | 100 | 100 | 9987 | 100 | 100 | 3151 |
| | 8 | 2 | 0 | 0 | 150000 | 100 | 100 | 2975 |
| | 9 | 2 | 100 | 100 | 15 | 100 | 100 | 1846 |
| | 10 | 2 | 100 | 100 | 13 | 100 | 100 | 2187 |

## 3.5 Conclusion

In this chapter we demonstrated how search based test data techniques can be used to generate test data at the lowest level of branch predicate. We adopted a rigorous empirical procedure to select the best parameter setting for SA. Using these parameters we generated the test data at the lowest level using a combine instrumentation/search strategy. This enabled us to target more practical coverage criteria such as MC/DC and multiple condition coverage. We also compared the performance of search based testing with random testing for these criteria. For easy targets, search based test data generation was more expensive in terms of number of fitness function evaluations, but mostly outperformed random testing for difficult branches. Random testing, as usually is the case, failed to find test data in cases where a narrow domain of search space contained the required input data. A reasonable strategy will be to hybridise search based techniques with random testing.

Chapter 4

# Application of Genetic Algorithms to Simulink Models

## 4.1 Introduction

Search based test data generation has been applied mostly to code. However, modern software engineering has seen increasing emphasis on the use of V&V at higher levels of abstraction, and the exploitation of the models that arise at higher levels for the purpose of test data generation [55, 93, 94]. One such design and modelling framework, used extensively in control system design is is MATLAB Simulink. Zhan and Clark [196] have demonstrated the application of search based test data generation techniques to Simulink models. Their work shows that code-oriented search based techniques can be applied in a very similar way to the Simulink models. The approach was evaluated with simulated annealing, which out performed random testing in most cases. This chapter proposes an extension of the above work suggesting the use of genetic algorithms. The results show that genetic algorithms can generate test data for Simulink models more efficiently.

The chapter is structured as follow. In the first section a short introduction to Simulink model is presented, followed by the application of search based test data generation technique to such models and the existing work. This is followed by the section which describes how genetic algorithms have been applied to Simulink models. The last section presents the results of experiments conducted to compare

the performance of simulated annealing and genetic algorithms.

## 4.2 Background

### 4.2.1 Simulink

Simulink is a software package from MathWorks [91] for modelling, simulating, and analysing dynamic systems. A Simulink model consists of two main elements: blocks and lines. Blocks are the functional units, used to generate, manipulate and output signals. Blocks are connected by lines that provide the mechanism to transfer signals across the connection. A block can be a parent container containing other blocks, each modeling a subsystem or sub-functionality.

The Simulink library consists of a large number of blocks. For test data generation using control flow coverage measures, we are interested in blocks that provide branching structures. Many such blocks can be found such as 'If' and 'Switch' blocks. An 'If' block can take any number of inputs and based on these inputs, construct output conditions. The conditions' out ports are connected to the respective 'If Action Subsystem'. The subsystem is executed when the corresponding output condition becomes true. For example, Figure 4.1 corresponds to the pseudocode segment in listing 4.1.

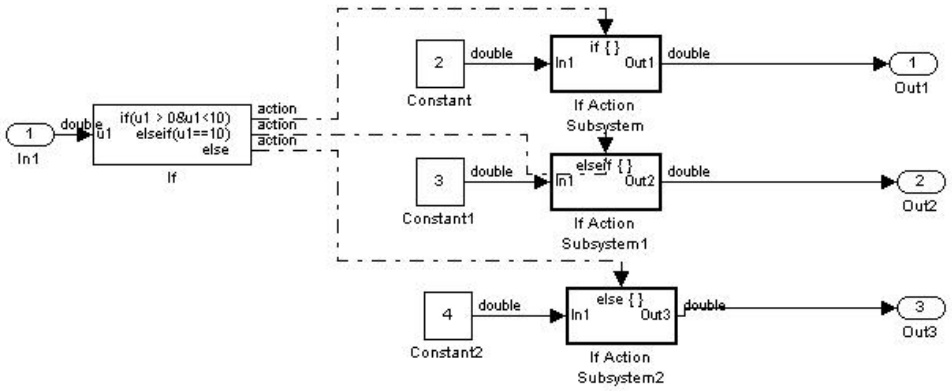


Figure 4.1: Simulink If Block

The 'Switch' block is the most commonly used branching block. It consists

```
1                   if  ((u1>0)&&(u1<10)){
2                       output (2)
3                       }
4                   else if (u1==10){
5                       output (3)
6                       }
7                   else {
8                       output (4)
9                       }
```

Listing 4.1: Pseudocode equivalent of Figure 4.1

of four 'ports': three 'in' ports and one 'out' port. The middle 'in' port provides a conditional value. If this value is greater than or equal to some 'threshold' value then the first 'in' port value is the output or vice-versa. Thus in Simulink a 'Switch' block can be a considered as a convenient way of writing the if-then-else structure.

The 'ConditionalOperator' and 'LogicalOperator' blocks provide conditional and logical operations in the model. Consider the example of the Quadratic model as shown in Figure 4.2 from Zhan [195]. The model demonstrates the above mentioned blocks. This model has three input variables; 'IN-A', 'IN-B' and 'IN-C'. The 'Product' block multiplies all its inputs. The 'Out' block gives the output. The model calculates if an equation of the form:

$$ax^2 + bx + c = 0$$

is both truly quadratic (i.e., has degree 2) and has real-valued (possibly identical) roots. If so, '1' is output; otherwise, the output is '-1'. Listing 4.2 provides the equivalent code listing of the model.

```
1     if(IN-A != 0) && (IN-B*IN-B)>=(4*IN-A*IN-C)
2         output(1)
3     else
4         output(-1);
```

Listing 4.2: Equivalent code listing of Quadratic model

Search based test data generation techniques measure how 'close' a proposed test input vector comes to achieving the required test goal. They do this by

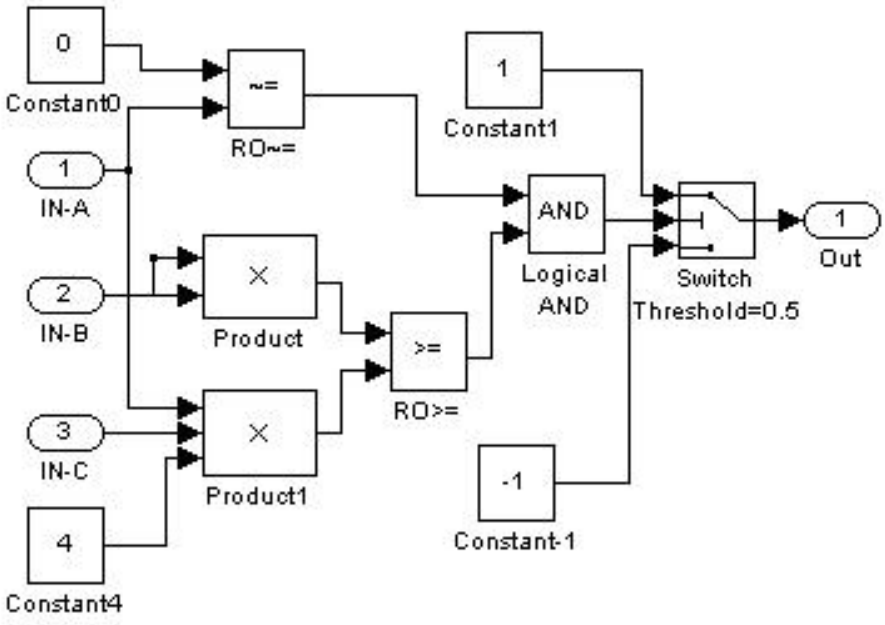


Figure 4.2: A Simulink Model for the Quadratic Equation.

monitoring the execution of the system description. Though useful feedback could in theory be provided by other means, execution of the system description is almost universally adapted. Our approach to obtaining feedback assumes that the model is executable (simulatable). In search based test data generation generally significant restrictions have been placed on data types that may participate in predicate expressions (e.g., relational expressions). The most common restriction is that predicates involve only scalar quantities and occasionally booleans. Some work [15, 103] has attempted to incorporate other data types. Our implementation of search based test data generation for MATLAB/Simulink restricts predicates to relational operators over numeric variables. (In MATLAB Simulink the default data type is double with boolean *true* and *false* represented by unsigned 8-bit integers 1 and 0 respectively).

### 4.2.2 Search Based Test Data Generation for Simulink Models

Simulink has been popularly used as a higher level system prototyping or design tool in many domains, including aerospace, automobile and electronics systems.

This facilitates investigation (e.g. for both verification and validation purposes as well as optimisation) of the system under consideration at an early stage of development. Code can then be generated either manually or automatically. Simulink is playing an increasingly important role in system engineering and the verification and validation of Simulink models is becoming vital to users. Search based test data generation (SBTDG) techniques have seen little application to Simulink models, which is surprising since the execution model of Simulink would seem to allow analogous SBTDG techniques to be applied as for code. Here we build on earlier work by Zhan and Clark [195]

### 4.2.3 Existing Work

Zhan and Clark [196] proposed search based techniques for generating test data for Simulink models. They successfully generated input test data for structural and mutation testing [197]. They also proposed a technique to address the state problem in search based testing [198] (extending code-based work by McMinn [121]

The search techniques that were applied for test data generation are random testing, which has been the choice for some of the existing tools for Simulink models, and simulated annealing. Various models were used for experiments. Results showed that SA was more efficient than random search in finding input test data. The work that we propose in this chapter is the extension of the above work. We suggest using GAs for test data generation as our results show that it may be more efficient in generating test data for Simulink models than SA.

### 4.2.4 GAs for Test Data Generation of Simulink Models

SA can be a useful technique for generating test data as suggested by existing work [176]. The existing work for Simulink models is in agreement with this. However, there are situations where SA may not be efficient in finding the test data i.e., where the search space is complex and the input domain is very large. In such situation there is a need to apply a more generalized search technique.

That is why, we see that GAs have been the most common choice in search based test data generation techniques.

We believe that GAs will also be more successful in case of Simulink models. However, before making any such claim, we need to have some foundation, either theoretical or empirical. We adapted the later approach for our work where we have attempted to make a comparative study of SA and GA, assuming that either of the two algorithms will perform better. We used Zhan's [195] existing prototype tool with some modification for SA. We also incorporated Genetic Algorithm and Direct Search toolbox (GADS) by The MathWorks, inc. [2] for GA in the same tool.

### 4.2.5 Fitness Function

In search based testing, fitness function plays a vital role in guiding search to achieve a goal. Since the application of search based test data generation technique to a Simulink model is very similar to code based programs, the same fitness function can be used. In the work by Zhan and Clark [196], the fitness function proposed by [174] is used with the modification proposed by [26]. Table 4.1 shows this fitness function. The same fitness function is used in this chapter for both SA and GA. In Simulink, as stated earlier, the output of all branches is calculated irrespective of branch selection. This gives the added advantage of calculating value at any point in the model. Probes can be inserted at appropriate locations to collect the data.

## 4.3 Experimentation

We performed two sets of Experiments. In the first set, we considered the 'all-paths-coverage' criterion of models as proposed in [196], where a path is made of an identified combination of switch blocks. Fulfillment of the structural adequacy criterion will require a test set to exercise all such combinations of switch predicates. For example, if a model contains two switch blocks $S_1$ and $S_2$, then the satisfaction of the above criterion will require finding test data for four 'paths',

Table 4.1: Cost Function

| Predicate | Value of Cost Function F |
|---|---|
| Boolean | if TRUE then 0, else K |
| $E_1 < E_2$ | if $E_1 - E_2 < 0$ then 0, else $E_1 - E_2 + K$ |
| $E_1 \leq E_2$ | if $E_1 - E_2 \leq 0$ then 0, else $E_1 - E_2 + K$ |
| $E_1 > E_2$ | if $E_2 - E_1 \leq 0$ then 0, else $E_2 - E_1 + K$ |
| $E_1 \geq E_2$ | if $E_1 - E_2 \leq 0$ then 0, else $E_2 - E_1 + K$ |
| $E_1 = E_2$ | if $abs(E_1 - E_2) = 0$ then 0, else $Abs(E_1 - E_2) + K$ |
| $E_1 \neq E_2$ | if $abs(E_1 - E_2) \neq 0$ then 0, else $K$ |
| $E_1 \vee E_2$ ($E_1$ unsatisfied, $E_2$ unsatisfied | (cost ($E_1$) x cost ($E_2$))/ (cost ($E_1$)+cost ($E_2$)) |
| $E_1 \vee E_2$ ($E_1$ unsatisfied, $E_2$ satisfied | 0 |
| $E_1 \vee E_2$ ($E_1$ satisfied, $E_2$ unsatisfied | 0 |
| $E_1 \vee E_2$ ($E_1$ satisfied, $E_2$ satisfied | 0 |
| $E_1 \wedge E_2$ ($E_1$ unsatisfied, $E_2$ unsatisfied | cost($E_1$)+cost($E_2$) |
| $E_1 \wedge E_2$ ($E_1$ unsatisfied, $E_2$ satisfied | cost($E_1$) |
| $E_1 \wedge E_2$ ($E_1$ satisfied, $E_2$ unsatisfied | cost($E_2$) |
| $E_1 \wedge E_2$ ($E_1$ satisfied, $E_2$ satisfied | 0 |

i.e. $S_1 - true\ S_2 - true$, $S_1 - true\ S_2 - false$, $S_1 - false\ S_2 - true$, and $S_1 - false\ S_2 - false$.

The 'all-path-coverage' criterion, however, in most cases, may not be practical, where the number of branching blocks is high. For example a model which contains 15 branching blocks may require to satisfy 32768 such 'paths', which may be computationally very expensive as well as impractical in a reasonable amount of time. Moreover, many such paths may be infeasible. Therefore, in the second set of experiments, we considered a more practical criterion of 'branch coverage' for three different Simulink blocks: *Relational*, *Conditional* and *Switch* [195]. The branch coverage criterion requires all conditional behaviours of the blocks to be executed at least once. For example, a *LogicalOperator* block has two conditional behaviours: being evaluated to 'TRUE' or 'FLASE'. Therefore, in the model in Figure 4.2, where there are four branching blocks ($RO{\sim}=$, $RO{>}=$, $LogicalAND$ and *Switch*), we have eight such branch coverage requirements.

The aims of the experiments are to assess:

- The degree to which a genetic algorithms based test data generation approach can generate test data for MATLAB Simulnk models.
- To compare genetic algorithms and simulated annealing based search based test data generation approaches in the context of MATLAB Simulink models.

### 4.3.1 Experimental Objects

For experiments, we used the models from the work of Zhan and Clark [196, 195]. These models have been summarised in Tables 4.2 and 4.3 and have been given in appendix B. Following is the description of the models. All the models, except Calc-Start-Progress are tested with double values in the range between -100 to 100 inclusive.

**SimpSw**

This is a simple model consisting of two 'Switch' blocks. The threshold value for 'Switch1' is 100 and for 'Switch2' is 50.

**Quadratic v1**

The model consists of 3 Switch blocks. The threshold value for 'Switch', 'Switch1' and 'Switch2' are 0.5, 1 and 1 respectively.

**RandMdl**

This model is more complex than the above two models. It consists of 4 switch blocks which gives a total of 16 combinations 'paths'. The threshold values for 'Switch1', 'Switch2', 'Switch3' and 'Switch4' are 8100, 1000, 0 and 8100 respectively. Switch2 and Switch4 provides an equivalent of 'nested if' structure in the code.

**CombineMdl**

CombineMdl is a combination of both Quardatic v1 and RandMdl. This makes the model more complex and difficult for test data generation process. It consists of 7 switch blocks which gives a total of 128 'paths'.

**Tiny**

Here is the description of the model given by the author in [195]. "This is a small and highly artificial program with a highly constrained solution domain. It works as follows. There are three inputs X, Y, Z. When the predicate expression

$((Y - Z) \times (Z - X) >= 1000) \wedge (Z \times Z >= 8950)$

is TRUE, the output will be $(X + Y)$. Otherwise, the output will be $(Z \times Z)$".

**Quadratic v2**

This model has been described in Section 4.2.

Table 4.2: Experimental objects1

| Model | No of Input Var | No of Switch Blocks | No of Paths |
|---|---|---|---|
| SmplSw | 2 | 2 | 4 |
| Quadratic v1 | 2 | 3 | 8 |
| RandMdl | 3 | 4 | 16 |
| CombineMdl | 5 | 7 | 128 |

Table 4.3: Experimental objects2

| Model | No of Input Var | No of Blocks | No of Branches |
|---|---|---|---|
| Tiny | 3 | 4 | 8 |
| Quadratic v2 | 3 | 4 | 8 |
| ClacStart | 3 | 25 | 50 |

**Calc-Start-Progress**

Following is the description from the author in [195] "This is also a subsystem of an engine controller system. It determines the overall progress of any starting process based on three signals indicating the status of various components of the system. Its three inputs are all integers, with a range of [1 .. 5000]. "

### 4.3.2 Experimental Setup

We used the following SA and GA configuration for both sets of experiments.

**Simulated Annealing Configuration :** We used the standard SA algorithm with the following parameters. The parameters values are based on the existing work in [195], where the parameters were optimized after a number of experiments.

- Move strategy: Fixed-strategy with a parameter of 0.02
- Geometric temperature decrease rate: 0.9
- Number of inner loop iterations: 100

- Maximum number of outer loop iterations: 300
- Stopping criteria: Either a solution is found or max number of iterations is reached.

**Genetic Algorithm Configuration :** We used the GADS toolbox [2], mostly, in its default configuration. Following is a summary of these parameters.

- Initial population size=100 (50 for SmplSW and Quadratic v1).
- Maximum number of generations: 300
- Selection: Stochastic uniform which chooses the parents using roulette wheel and uniform sampling [2].
- Elite count: 2
- Crossover rate: 0.8
- Mutation function: Gaussian which creates the mutated children using Gaussian distribution (scale=.5, shrink=0.75).
- Stopping criteria: Either a solution is found, a stall limit of 100 generations is reached, or the maximum number of generations limit is reached.

### 4.3.3 Analysis

Tables 4.2 and 4.3 show the number of paths (combinations) and branches for the models. For analysis we did not consider trivial combinations. We defined a trivial path or branch as the one for which both the algorithms found required data easily, i.e., SA took less than 100 executions whereas GA took 3 or fewer generations to find the test data in all runs. Each algorithm was run 30 times for all the models to obtain statistically significant results.

Our null hypothesis is that *neither of the algorithms is better than the other.* The two variables we tested our hypothesis on, are success rate and number of

Table 4.4: GA and SA comparison for Quadratic v1 Model

| No | Combinations | Mean no of Evaluations | | Success rate per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 1 | 1 1 1 | 50 | 69 | 30 | 30 |
| 2 | 1 1-1 | 431 | 2164 | 30 | 30 |
| 3 | 1-1 1 | 697 | 576 | 25 | 30 |
| 4 | 1-1-1 | 50 | 48 | 30 | 30 |
| 5 | -1 1 1 | 198 | 258 | 30 | 30 |
| 6 | -1 1-1 | 1127 | 2287 | 13 | 30 |
| 7 | -1-1 1 | 447 | 344 | 30 | 30 |
| 8 | -1-1-1 | 253 | 212 | 30 | 30 |

executions each algorithm took to find the input test data. Tables 4.4 to 4.9 give the results of experiments.

Both algorithms achieved 100% coverage for all the branches in both experiments. However, when compared for success rate, in experiment 1, GA performed better than SA. GA achieved a much higher success rate in more complex models. In 'Quadratic v1' SA showed better success rate for two of the combinations. In experiment 2, the performance of both the algorithms was not much different.

In experiment 1, GA required less number of fitness evaluations to find a solution for simple models but SA performed much better than the GA for more complex models. In case of experiment 2, however, GA performed much better for more complex models.

Table 4.5: GA and SA Comparison for Random Model

| No | Combinations | Mean no of Evaluations | | Success rate per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 1 | 1 1 1 1 | 2995 | 402 | 21 | 5 |
| 2 | 1 1 1-1 | 1250 | 716 | 10 | 9 |
| 3 | 1 1-1 1 | 100 | 72 | 30 | 8 |
| 4 | 1 1-1-1 | 100 | 88 | 30 | 7 |
| 5 | 1-1 1 1 | 100 | 288 | 30 | 16 |
| 6 | 1-1 1-1 | 100 | 248 | 27 | 8 |
| 7 | 1-1-1 1 | 100 | 175 | 30 | 23 |
| 8 | 1-1-1-1 | 100 | 245 | 30 | 30 |
| 9 | -1 1 1 1 | 2233 | 2882 | 18 | 11 |
| 10 | -1 1 1-1 | 2554 | 453 | 24 | 10 |
| 11 | -1 1-1 1 | 100 | 348 | 30 | 11 |
| 12 | -1 1-1-1 | 100 | 1023 | 30 | 9 |
| 13 | -1-1 1 1 | 100 | 84 | 30 | 12 |
| 14 | -1-1 1-1 | 100 | 294 | 30 | 20 |
| 15 | -1-1-1 1 | 100 | 168 | 30 | 30 |
| 16 | -1-1-1-1 | 100 | 183 | 30 | 30 |

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 1 | 1 1 1 1 1 1 1 | 3060 | 480 | 23 | 6 |
| 2 | 1 1 1 1 1 1-1 | 1823 | 337 | 22 | 10 |
| 3 | 1 1 1 1 1-1 1 | 4567 | 522 | 21 | 4 |
| 4 | 1 1 1 1 1-1-1 | 4831 | 811 | 16 | 2 |
| 5 | 1 1 1 1-1 1 1 | 1600 | 624 | 10 | 3 |
| 6 | 1 1 1 1-1 1-1 | 1127 | 1014 | 11 | 2 |
| 7 | 1 1 1 1-1-1 1 | 2400 | 590 | 8 | 5 |
| 8 | 1 1 1 1-1-1-1 | 2789 | 629 | 9 | 7 |
| 9 | 1 1 1-1 1 1-1 | 183 | 264 | 30 | 9 |

continued on next page

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 10 | 1 1 1-1 1-1 1 | 1308 | 304 | 25 | 5 |
| 11 | 1 1 1-1 1-1-1 | 1479 | 327 | 28 | 10 |
| 12 | 1 1 1-1-1 1 1 | 100 | 276 | 30 | 25 |
| 13 | 1 1 1-1-1 1 1 | 103 | 351 | 29 | 8 |
| 14 | 1 1 1-1-1 1-1 | 107 | 245 | 30 | 9 |
| 15 | 1 1 1-1-1-1 1 | 1104 | 331 | 26 | 10 |
| 16 | 1 1 1-1-1-1-1 | 1138 | 452 | 26 | 4 |
| 17 | 1 1-1 1 1 1 1 | 114 | 402 | 29 | 19 |
| 18 | 1 1-1 1 1 1-1 | 117 | 559 | 29 | 14 |
| 19 | 1 1-1 1 1-1 1 | 983 | 551 | 29 | 19 |
| 20 | 1 1-1 1 1-1-1 | 1080 | 624 | 26 | 17 |
| 21 | 1 1-1 1-1 1 1 | 144 | 415 | 27 | 9 |
| 22 | 1 1-1 1-1 1-1 | 154 | 2828 | 28 | 13 |
| 23 | 1 1-1 1-1-1 1 | 778 | 1235 | 27 | 9 |
| 24 | 1 1-1 1-1-1-1 | 1341 | 296 | 27 | 8 |
| 25 | 1 1-1-1 1 1 1 | 100 | 276 | 30 | 25 |
| 26 | 1 1-1-1 1 1-1 | 100 | 325 | 30 | 22 |
| 27 | 1 1-1-1 1-1 1 | 770 | 1494 | 30 | 26 |
| 28 | 1 1-1-1 1-1-1 | 536 | 370 | 30 | 22 |
| 29 | 1 1-1-1-1 1 1 | 100 | 355 | 30 | 30 |
| 30 | 1 1-1-1-1 1-1 | 100 | 384 | 30 | 30 |
| 31 | 1 1-1-1-1-1 1 | 873 | 485 | 30 | 30 |
| 32 | 1 1-1-1-1-1-1 | 741 | 576 | 29 | 29 |
| 33 | 1-1 1 1 1 1 1 | 1817 | 4393 | 12 | 15 |

continued on next page

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 34 | 1-1 1 1 1 1-1 | 3910 | 4178 | 11 | 12 |
| 35 | 1-1 1 1 1-1 1 | 3342 | 6404 | 12 | 14 |
| 36 | 1-1 1 1 1-1-1 | 3981 | 2846 | 16 | 17 |
| 37 | 1-1 1 1-1 1 1 | 3017 | 5231 | 24 | 11 |
| 38 | 1-1 1 1-1 1-1 | 2738 | 2946 | 24 | 14 |
| 39 | 1-1 1 1-1-1 1 | 5000 | 3037 | 17 | 10 |
| 40 | 1-1 1 1-1-1-1 | 5353 | 2241 | 19 | 12 |
| 41 | 1-1 1-1 1 1 1 | 120 | 2443 | 30 | 16 |
| 42 | 1-1 1-1 1 1-1 | 100 | 1606 | 30 | 18 |
| 43 | 1-1 1-1 1-1 1 | 900 | 2595 | 26 | 25 |
| 44 | 1-1 1-1 1-1-1 | 1144 | 1496 | 27 | 14 |
| 45 | 1-1 1-1-1 1 1 | 100 | 2378 | 30 | 17 |
| 46 | 1-1 1-1-1 1-1 | 113 | 2192 | 30 | 14 |
| 47 | 1-1 1-1-1-1 1 | 1155 | 1398 | 29 | 18 |
| 48 | 1-1 1-1-1-1-1 | 1046 | 3002 | 28 | 15 |
| 49 | 1-1-1 1 1 1 1 | 150 | 292 | 30 | 14 |
| 50 | 1-1-1 1 1 1-1 | 220 | 421 | 29 | 13 |
| 51 | 1-1 1-1-1-1-1 | 980 | 484 | 25 | 10 |
| 52 | 1-1-1 1 1-1-1 | 1166 | 406 | 29 | 16 |
| 53 | 1-1-1 1-1 1 1 | 110 | 394 | 30 | 21 |
| 54 | 1-1-1 1-1 1-1 | 100 | 329 | 29 | 20 |
| 55 | 1-1-1 1-1-1 1 | 1032 | 420 | 28 | 17 |
| 56 | 1-1-1 1-1-1-1 | 1183 | 460 | 24 | 17 |
| 57 | 1-1-1-1 1 1 1 | 100 | 352 | 30 | 30 |

continued on next page

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 58 | 1-1-1-1 1 1-1 | 100 | 332 | 30 | 30 |
| 59 | 1-1-1-1 1-1 1 | 443 | 472 | 30 | 30 |
| 60 | 1-1-1-1 1-1-1 | 570 | 514 | 30 | 30 |
| 61 | 1-1-1-1-1 1 1 | 100 | 276 | 30 | 30 |
| 62 | 1-1-1-1-1 1-1 | 100 | 251 | 30 | 30 |
| 63 | 1-1-1-1-1-1 1 | 466 | 459 | 30 | 30 |
| 64 | 1-1-1-1-1-1-1 | 536 | 572 | 30 | 30 |
| 65 | -1 1 1 1 1 1 1 | 3109 | 444 | 21 | 3 |
| 66 | -1 1 1 1 1 1-1 | 2804 | 496 | 23 | 7 |
| 67 | -11 1 1 1-1 1 | 4010 | 554 | 19 | 6 |
| 68 | -1 1 1 1 1-1-1 | 3250 | 739 | 14 | 6 |
| 69 | -1 1 1 1-1 1 1 | 1713 | 455 | 15 | 7 |
| 70 | -1 1 1 1-1 1-1 | 1900 | 611 | 10 | 5 |
| 72 | -1 1 1 1-1-1 1 | 2789 | 499 | 9 | 4 |
| 73 | -1 1 1 1-1-1-1 | 2567 | 734 | 9 | 3 |
| 74 | -1 1 1-1 1 1 1 | 272 | 242 | 29 | 9 |
| 75 | -1 1 1-1 1 1-1 | 227 | 301 | 29 | 4 |
| 76 | -1 1 1-1 1-1 1 | 1318 | 313 | 28 | 11 |
| 77 | -1 1 1-1 1-1-1 | 1748 | 316 | 27 | 3 |
| 78 | -1 1 1-1-1 1 1 | 130 | 212 | 30 | 8 |
| 79 | -1 1 1-1-1 1-1 | 157 | 248 | 30 | 5 |
| 80 | -1 1 1-1-1-1 1 | 881 | 306 | 27 | 8 |
| 81 | -1 1 1-1-1-1-1 | 1329 | 324 | 24 | 3 |
| 82 | -1 1-1 1 1 1 1 | 146 | 579 | 24 | 21 |

continued on next page

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 83 | -1 1-1 1 1 1-1 | 169 | 2122 | 29 | 20 |
| 84 | -1 1-1 1 1-1 1 | 1231 | 563 | 26 | 21 |
| 85 | -1 1-1 1 1-1-1 | 1252 | 585 | 25 | 19 |
| 86 | -1 1-1 1-1 1 1 | 168 | 462 | 28 | 14 |
| 87 | -1 1-1 1-1 1-1 | 165 | 337 | 26 | 11 |
| 88 | -1 1-1 1-1-1 1 | 1596 | 429 | 26 | 4 |
| 89 | -1 1-1 1-1-1-1 | 1652 | 493 | 21 | 7 |
| 90 | -1 1-1-1 1 1 1 | 103 | 328 | 30 | 21 |
| 91 | -1 1-1-1 1 1-1 | 103 | 305 | 30 | 18 |
| 92 | -1 1-1-1 1-1 1 | 773 | 500 | 30 | 25 |
| 93 | -1 1-1-1 1-1-1 | 1203 | 432 | 30 | 26 |
| 94 | -1 1-1-1-1 1-1 | 107 | 390 | 30 | 30 |
| 95 | -1 1-1-1-1-1 1 | 1573 | 579 | 30 | 30 |
| 96 | -1 1-1-1-1-1-1 | 1777 | 550 | 30 | 30 |
| 97 | -1-1 1 1 1 1 1 | 2372 | 4402 | 18 | 15 |
| 98 | -1-1 1 1 1 1-1 | 4057 | 3572 | 14 | 19 |
| 99 | -1-1 1 1 1-1 1 | 5371 | 6617 | 7 | 10 |
| 100 | -1-1 1 1 1-1-1 | 5364 | 6956 | 11 | 16 |
| 101 | -1-1 1 1-1 1 1 | 3656 | 4339 | 18 | 14 |
| 102 | -1-1 1 1-1 1-1 | 3400 | 3852 | 18 | 9 |
| 103 | -1-1 1 1-1-1 1 | 4258 | 1454 | 19 | 12 |
| 104 | -1-1 1 1-1-1-1 | 3264 | 1064 | 22 | 10 |
| 105 | -1-1 1-1 1 1 1 | 130 | 338 | 30 | 13 |
| 106 | -1-1 1-1 1 1-1 | 128 | 2139 | 29 | 15 |

continued on next page

Table 4.6: Combine Model GA and SA Comparison

| No | Combination | Mean no of Evaluations | | Success per 30 runs | |
|---|---|---|---|---|---|
| | | GA | SA | GA | SA |
| 107 | -1-1 1-1 1-1 1 | 1115 | 1727 | 27 | 18 |
| 108 | -1-1 1-1 1-1-1 | 1104 | 2847 | 23 | 12 |
| 109 | -1-1 1-1-1 1 1 | 138 | 647 | 26 | 15 |
| 110 | -1-1 1-1-1 1-1 | 237 | 296 | 30 | 13 |
| 111 | -1-1 1-1-1-1 1 | 1389 | 2334 | 27 | 14 |
| 112 | -1-1 1-1-1-1-1 | 1173 | 1681 | 26 | 17 |
| 113 | -1-1-1 1 1 1 1 | 182 | 622 | 27 | 9 |
| 114 | -1-1-1 1 1 1-1 | 319 | 2083 | 27 | 18 |
| 115 | -1-1-1 1 1-1 1 | 896 | 515 | 25 | 17 |
| 116 | -1-1-1 1 1-1-1 | 871 | 521 | 24 | 13 |
| 117 | -1-1-1 1-1 1 1 | 164 | 531 | 28 | 13 |
| 118 | -1-1-1 1-1 1-1 | 293 | 448 | 28 | 17 |
| 119 | -1-1-1 1-1-1 1 | 986 | 452 | 21 | 18 |
| 120 | -1-1-1 1-1-1-1 | 1030 | 424 | 27 | 16 |
| 121 | -1-1-1-1 1 1 1 | 103 | 339 | 30 | 30 |
| 122 | -1-1-1-1 1 1-1 | 106 | 485 | 30 | 30 |
| 123 | -1-1-1-1 1-1 1 | 830 | 587 | 30 | 30 |
| 124 | -1-1-1-1 1-1-1 | 736 | 457 | 30 | 30 |
| 125 | -1-1-1-1-1 1 1 | 100 | 414 | 30 | 30 |
| 126 | -1-1-1-1-1 1-1 | 100 | 509 | 30 | 30 |
| 127 | -1-1-1-1-1-1 1 | 769 | 342 | 30 | 30 |
| 128 | -1-1-1-1-1-1-1 | 577 | 471 | 30 | 30 |

Table 4.7: GA and SA comparison for Tiny for branch coverage

| Branch No | Mean no of Evaluations | | Success rate | |
|---|---|---|---|---|
| | GA | SA | GA | SA |
| 3 | 4596 | 1343 | 24 | 30 |
| 5 | 100 | 151 | 30 | 30 |
| 7 | 103 | 165 | 30 | 30 |

Table 4.8: GA and SA comparison for Quadratic v2 for branch coverage

| Branch No | Mean no of evaluations | | Success rate | |
|---|---|---|---|---|
| | GA | SA | GA | SA |
| 2 | 273 | 2861 | 30 | 30 |
| 4 | 100 | 1953 | 30 | 30 |
| 6 | 100 | 154 | 30 | 30 |
| 8 | 2417 | 11187 | 30 | 27 |

The results give GA an edge over SA. However, when compared for the means of the fitness function evaluations, when input data was found, we found that the number of paths/branches for which SA performed better than GA were mostly the same as the number of paths/branches for which GA performed better as shown in the last two columns of Table 4.10. However, in experiment 2, GA outperformed SA. Still, however, this does not give us much information about statistically significant difference between the two algorithms in terms of this measure of performance. To find out which algorithm performed significantly better than the other when test data was found, we further conducted the Mann-Whitney non parametric test for the number of fitness function evaluations. Table 4.11 summarizes the results of Mann-Whitney test for both experiments. Column 2 in Table 4.11 gives the total number of paths or branches considered for analysis. Column 3 gives the number of paths of respective models for which SA performed significantly better than GA. Whereas column 4 gives the same for GA. Column 5 gives us the number of branches for which there wasn't a significant difference between the two algorithms.

From the analysis results we can see that GA performed slightly better than SA when compared for the number of execution it took to find test data. However, for a significant amount of time there wasn't any statistically significant difference

Table 4.9: GA and SA comparison for CalcStart for branch coverage

| Branch No | Mean | | Success rate per 30 runs | |
|---|---|---|---|---|
| | GA | SA | GA | SA |
| 1 | 2270 | 1459 | 30 | 30 |
| 3 | 2350 | 1774 | 30 | 30 |
| 5 | 1000 | 823 | 30 | 30 |
| 7 | 1920 | 2210 | 30 | 30 |
| 9 | 2170 | 1899 | 30 | 30 |
| 11 | 2560 | 2107 | 30 | 30 |
| 13 | 2300 | 2743 | 30 | 30 |
| 15 | 2040 | 1717 | 30 | 30 |
| 17 | 380 | 931 | 30 | 30 |
| 19 | 3540 | 14987 | 30 | 15 |
| 21 | 2390 | 2269 | 30 | 30 |
| 23 | 2380 | 2284 | 30 | 30 |
| 25 | 2090 | 1981 | 30 | 30 |
| 27 | 2310 | 1420 | 30 | 30 |
| 29 | 310 | 924 | 30 | 30 |
| 31 | 2320 | 2231 | 30 | 30 |
| 34 | 100 | 1055 | 30 | 30 |
| 35 | 2380 | 1684 | 30 | 30 |
| 37 | 2389 | 2016 | 30 | 30 |
| 39 | 2070 | 1795 | 30 | 30 |
| 41 | 850 | 755 | 30 | 30 |
| 43 | 820 | 2238 | 30 | 30 |
| 45 | 1680 | 1949 | 30 | 30 |
| 47 | 2250 | 2156 | 30 | 30 |
| 49 | 100 | 497 | 30 | 30 |

Table 4.10: Results of Experiments

| | Coverage | | Mean success rate | | Comparison | |
|---|---|---|---|---|---|---|
| | SA | GA | SA | GA | SA | GA |
| SmplSw | 2/2 | 2/2 | 30 | 30 | 0 | 1 |
| Quadratic v1 | 8/8 | 8/8 | 30 | 28 | 4 | 4 |
| RandMdl | 16/16 | 16/16 | 15 | 25 | 8 | 8 |
| CombineMdl | 128/128 | 128/128 | 17 | 27 | 63 | 63 |
| Quadratic v2 | 8/8 | 8/8 | 29.625 | 30 | 0 | 4 |
| Tiny | 8/8 | 8/8 | 28.5 | 30 | 2 | 2 |
| Calc-Start-Progress | 50/50 | 50/50 | 29.7 | 30 | 8 | 17 |

between the two.

## 4.4 Conclusion

In this chapter we presented the work for an empirical comparison of simulated annealing and genetic algorithms for Simulink models. To the best of our knowledge, this is the first study of the type for the Simulink models. Surprisingly, we also do not see much work of the kind for code based systems. The only work that came to our knowledge is from Nashat et al [111] and Tracey et al [176]. In [111], results show that SA performed better than GA and, therefore, they suggested using SA for generating test data for path testing. In the work of Tracey et al. SA was more efficient for simpler code segments but performed similar to the GA for more complex programs. However, our results show contrary to this. GA performed slightly better than SA, when compared for the fitness function evaluations. But for the most part, there was not much 'statistically significant' difference between the two. However, when it comes to the success rate, the performance of GA was much better than SA, thus making it a more attractive choice for the test data generation for Simulink models.

SBTDG is the most addressed field within search-based software engineering.

Table 4.11: Mann-Whitney Analysis of Experiments

| Model Analyzed | No of Paths/ Branches Analyzed | SA | GA | No Difference |
|---|---|---|---|---|
| Smplsw | 1 | 0 | 1 | 0 |
| Quadratic v1 | 8 | 0 | 3 | 5 |
| RandMdl | 15 | 3 | 3 | 9 |
| CombineMdl | 126 | 34 | 42 | 50 |
| Tiny | 4 | 2 | 0 | 2 |
| Quadratic v2 | 4 | 0 | 3 | 1 |
| Calc-Start-Progress | 25 | 6 | 6 | 13 |

Indeed, it could be said that SBSE grew out of work in SBTDG. However, we believe that the full potential of SBTDG will only be revealed when its techniques can be routinely applied across system descriptions of varying degrees of abstraction. There has been a great deal of work at the code level, and work generating test data from specifications, but surprisingly little in the middle ground of design. Our work here provides an initial comparison of search approaches to systems expressed in one such design notation and does so with experimental rigour. Longer term, it is clear that a rigorous mapping is needed between system complexity (however measured) and efficacy of the various search techniques. This is a valuable long-term strategic goal and which require further research.

Chapter 5

# Program Stretching

## 5.1 Introduction

As discussed in the previous chapter, search based techniques have been successfully applied to generate test data for many programs. Current approaches to search based test data generation have explored the use of various optimisation techniques and various fitness functions to find test data satisfying some particular goal. The approaches tried have seen considerable success, but there are also clear limitations.

Even with the predicate structures involving Boolean relational operators, some targets will be difficult to achieve. It may be because of the very small input domain of data satisfying the goal or because the program structures are not very amenable to the search process. For example, handling Boolean flag variables is hard. In such cases one approach has been to transform the program to an equivalent version more suited to search based approaches [84, 76, 119].

In this chapter we also propose a transformation-based approach called "program stretching". The approach has been proposed for efficient input test data generation of hard-to-reach targets using search based techniques. The chapter begins with a brief introduction to the program transformation followed by a review of its application to search based software testing. This will be followed by a description of our proposed approach: "program stretching". Our approach is then evaluated through a number of experiments in the experimental and evalua-

tion section.

## 5.2 Program Transformation

In the literature, the term "program transformation" applies to a variety of processes. The common theme is that a source program is mapped to a different but related system description. The levels of description may be at different levels of abstraction. Thus, a compiler can be said to transform a source program to binary (with binary obviously being at a lower level of abstraction than typical source code). This is often called 'vertical transformation'. Production of Java Byte code from a Java source program is similarly a vertical transformation. There are opportunities to transform from higher level system descriptions too. Thus, tools that support MATLAB SIMULINK models typically provide a facility to produce an executable simulation of the model by producing a C-code (or similar) implementation thereof [4, 6].

Other interpretations of program transformation refer to the manipulation of the source program into a closely related source program, with the goal of improving some aspect of interest. This is often referred to as "rephrasing" and "translation" (though the latter is also used in compiler construction to denote some vertical transformations).

For example, a compiler may (at various levels) make small changes to a program with the aim of improving efficiency. A statement within a loop may be "hoisted" outside the loop if it is unaffected by the function of the loop body. A compiler may similarly replace an expression of the form $(x = *16)$ by $(x << 4)$ (multiplication by 16 is the same as left shift by 4) when operating on integers. It may also alter the order of low level instruction execution (without altering the semantics of the instruction sequence overall, but allowing lower level optimisations, e.g., on register use, to take place). The term "peephole optimisation" is used to refer to a process of applying a suite of localised transformations by compilers. (Localised here means that the changes take place only on closely situated instruction sequences.)

Other transformations are possible. Thus small changes may preserve the semantics of a program but make it more comprehensible (according to some metric of comprehensibility). Such transformations may be very important for maintenance purposes. For example, a variety of tools exist for imposing 'structure' on poorly written 'spaghetti code'.

A common theme in all the above examples is that the transformations preserve functional semantics of the program. For some purposes this may not be necessary. For example, Harman et al have proposed *Testability Transformation* ($TeTra$) to deal with some problematic areas in search based software test data generation. Since then it has been applied successfully to the flag problem [76], nested predicates [119], and the state problem [93]. Harman [78] suggested other possible application areas in software engineering. Section 5.3 describes *testability transformation* in more detail.

### 5.2.1 Model Transformation

Another form of transformation practiced in software engineering community is model transformation. Models are used to address the size and complexity of a system by describing different aspects of it at various levels of abstraction. Many models are usually required for this purpose.

Often transformation is carried out to derive one model from another. This not only helps to extract a different aspect of the system but also keeps the model consistent. There is even more emphasis on model transformation in the current model driven development (MDD) approaches. Model transformation is one of the core elements of the set of standards defined by MDA (model driven architecture), an initiative of object management group (OMG) [5] to standarise MDD. OMG has defined a standard language QVT MOF 2.0 for model transformation [9]. An overview of QVT can be found in [102].

There are many approaches to model transformation. Sendall and Kozaczynski [171] categorise it in to three broad categories: *direct model manipulation*; *intermediate representation*; and *transformation language support*. In *direct model manipulation*, procedural APIs are used to manipulate internal model representa-

tion. In *intermediate representation*, models are exported in a standard form, such as XML and then transformed. *Transformation language support*, as the name suggests, defines languages for model transformation. We also find many other classifications. For a complete overview readers are referred to [42, 170, 124, 126].

## 5.3 Testability Transformation

Testability Transformation is the rephrasing of a program in order to make it more amenable to search. It is different from traditional transformation in two ways [81]:

1. In traditional transformation, the original program is discarded after transforming it into a target program. It is not so in Testability Transformation. Once the test data is found, the transformed program is discarded. Thus the transformed program is just a "means to an end".

2. It is not required for the transformation process to preserve the whole semantics of a program. Only those semantics are required which may be essential for test data generation.

This makes testability transformation very flexible. One need not worry much about proving functional equivalence or proof of correctness of the target program [79]. It is just required to be structured enough to help achieve the objective of test data generation.

### Definitions

For the sake of completeness, in this section, the definitions of Testability Transformation are presented, as given by [81].

**Definition 5.3.1 (Testing-Oriented Transformation)** *Let $P$ be a set of programs and $C$ be a set of test adequacy criteria. A program transformation is a partial function in $P \rightarrow P$. A* ***Testing-Oriented Transformation*** *is a partial function in $(P \times C) \rightarrow (P \times C)$.*

**Definition 5.3.2 (Testability Transformation)** *A Testing-Oriented Transformation $\tau$, is a* ***Testability Transformation*** *iff for all programs p, and criteria c, if $\tau(p, c) = (p', c')$ then for all test sets $T$, $T$ is adequate for $p$ according to $c$ if $T$ is adequate for $p'$ according to $c'$.*

For some criterion c, a c-preserving testability transformation guarantees that the transformed program is suitable for testing with respect to the original criterion.

**Definition 5.3.3 (c−Preserving Testability Transformation)** *Let n be a testability transformation. If, for some criterion, c, for all programs p, there exists some program $p'$ such that $\tau(p, c) = (p', c)$ called a* ***c−Preserving Testability Transformation****.*

One thing to be noted here is that all search based dynamic test data generation techniques are essentially transformation-based. The test data generation process is not run on the original program. Rather the program is instrumented with code elements in order to get data to be used in the cost function calculation. The instrumented or transformed version is thrown away once data is found. However, testability transformation adds, before conversion into the final version, an additional middle layer in the transformation process for test data generation.

### 5.3.1 Survey

Testability transformation was initially applied to the flag problem by Harman et al [76] for evolutionary test data generation. The flag problem has been introduced in Chapter 2. In their approach, Harman et al transformed the original program to an intermediate 'flag-free' program preserving the branches of the original program. Thus data can be generated for the required coverage criterion or a stronger criterion. Consider Example 5.3.1 [117]. After transformation the flag-use is replaced by the right hand side of its assignment expression. This provides more guidance to the search process improving both the performance of evolutionary test data generation and the adequacy level of the test data so generated.

**Example 5.3.1** *Testability Transformation applied to Flag Problem.*

| | |
|---|---|
| `{`<br>`flag=(a>b); ____ (i)`<br>`.......`<br>`if (flag)  ____ (ii)`<br>`{do something...}`<br>`}` | `{`<br><br>`flag=(a>b); ____ (i)`<br>`.......`<br>`if (a>b)  ____ (ii)`<br>`{do something...}`<br>`}` |
| (a)Original | (b)Transformed. |

In the example above we presented a simple case of Testability transformation. In their work, Harman et al. applied it to more complex cases. They defined five level of complexity of flag assignment [81] and provided algorithms to find test data. The fifth level consists of loop assigned flag, which were investigated in [79]. Wappler et al. [183] extended the work to function assigned flags i.e., the cases where the flag value is assigned the return value of a function. They suggested a combined testability transformation instrumentation strategy.

McMinn et al. [119] proposed a testability transformation approach to the nested search targets. In the traditional approach to search based test data generation, the search does not have knowledge of a branch until it reaches it. It can cause problems in the case of nested structures especially in cases where the branch variables are also used in the preceding branches. For example consider Example 5.3.2. The traditional approach must satisfy sub goal at branch (i) first. After satisfying it, if the value of $b$ is less than 100, the search would set up another value of $b$. This will cause the condition at branch (i) to became false and satisfaction of this will again become the focus of the search. Thus all the information previously available at branch (ii) is lost. McMinn et al's approach includes a transformation strategy that removes all the control-dependent decisions up to the target, but preserves their information in extra variables. This has been shown in Example 5.3.2. The advantage of this strategy is that it gives a smooth downward slope to the fitness curve, thus eliminating the requirement

to satisfy the preceding branches on which the target is control dependent. Case studies were performed on two small programs which showed an improvement factor of more than two in terms of efficiency in the search process. However, the technique still faces challenges posed by some common program constructs: loops, arrays and pointers.

Kalaji et al. [93] proposed a testability transformation approach to tackle special cases of the state problem. The state problem has been introduced in Chapter 2. They considered four cases depending upon type of assignment to the state variable. They defined two types of functions i.e., 'affecting' and 'affected-by'. Affected-by function is always data dependent on affecting function. In addition the affected-by function may also be control dependent on affecting function if there is some condition in affecting function that affect the assignment of state variables. Their transformation strategy included combining both above functions into a single function which contains the target condition directly dependent on input variables, thus giving a smooth downward slope to the fitness curve.

**Example 5.3.2** *Testability Transformation applied to nested predicates.*

| | |
|---|---|
| `{`<br><br>`if (a==b) ____ (i)`<br>`  {`<br>`  if (b>100)  ____ (ii)`<br>`{target}`<br>`}` | `{`<br><br>`  double dist=0`<br>`dist+=branch_distance(a==b); ____ (i)`<br>`.......`<br>`dist+=branch_distance(b>100);  ____ (ii)`<br><br>`if(dist==0.0)`<br>`{target}`<br>`}` |
| (a)Example 5.3.2 Original | (b)Example 5.3.2 Transformed. |

The above survey shows that testability transformation can be a useful tool to solve many problems in search based test data generation. The reader is referred to Harman [78] for a more extensive discussion on the applicability of testability transformation to other problems in search based software engineering.

## 5.4 Program Stretching

Program stretching is a program transformation technique which has been proposed to find test data for 'hard' targets in a program. The notion of 'hard' can have many meanings. The target can be hard because of the limitation of search based approaches or it can be hard because of structural complexity. We consider 'hard' in the latter sense.

Program stretching is essentially a ***c−Preserving Testability Transformation***, which means that we preserve the adequacy criterion during the transformation process. We 'stretch' the difficult branches thus making them easier to cover. This can be achieved by generating transformed programs by adding auxiliary variables to the predicates in the difficult branches. To further clarify the idea consider the following example.

**Example 5.4.1** *Program Stretching*

$if\ (expr1 < expr2)$ .....*(I)*
{
*statements*
}
*else*
{
*statements*
}

$if((expr3 > expr4) \wedge (expr5 < expr6))$.....*(II)*
{
*statements*
}
*else*
{
*statements*

}

*Lets suppose that (II) is the branch that SBTDG seems unable to cover. We now add additional input variables, i.e, var1 and var2 to it as shown below.*

$if\ (expr1 < expr2)$ .....*(I)*
{
*statements*
}
*else*
{
*statements*
}

$if\ ((expr3 + var1) > expr4 \wedge (expr5 <$
$(expr6 + var2))$.....*(II)*
{
*statements*
}
*else*
{
*statements*
}

Initially we set the values of $var1$ and $var2$ reasonably large so that it is easy (even trivial) to find test data such that $(expr3 + var1) > (expr4)$ and $expr5 < (expr6 + var2)$. The ranges of additional variable values define a set of "stretched" programs. Our search now proceeds over the set of such stretched programs and test inputs for them. The search trajectory comprises a sequence of pairs $\langle (prog_1, td_1), (prog_2, td_2), \ldots, (prog_{final}, td_{final}) \rangle$. The aim is to end up with

an "unstretched" program $prog_{final}$ (with all auxiliary variables set to 0) and test data $td_{final}$ that satisfies the required constraints. Essentially, the search space comprises the original test data input space combined with the set of all variable assignments to auxiliary variables. Although we know the desired eventual value of each auxiliary variable (i.e., 0) allowing it to take positive intermediate values can facilitate the solution of the overall problem. Essentially we find test data satisfying our goal for a highly stretched program, and evolve the test input data and auxiliary variables together to achieve the original aim. This requires a very simple modification to the existing cost function as shown in the following section. Program stretching is a one off process. It should not incur any more cost than a simple transformation step. The computation cost for search may be more as the search needs to bring the auxiliary variables' values to 0 and must evaluate this aspect of fitness at each evaluation. This, however, may not always be the case as shown by the experimental results for more complex code based program.

### 5.4.1 Cost Function

We have already discussed the cost function on page 66. This function is given below as.

$$f(x)_{branch} = f(x)_{branch_c} + KN$$

Where $K$ is a constant, $branch_c$ is the current branch and $N$ is the number of uncovered branches of the path to be satisfied.

We need also to incorporate the effect of auxiliary variables. For this purpose we add a new term to the fitness function, which is the summation of all the new variables. i.e.,

$$f(x)_{total} = f(x)_{branch} + \sum_{i=0}^{n} abs(v_i)$$

where $v_i$ is the $i$th auxiliary variable.

We then bring down $f(x)_{total}$ to zero by decreasing the value of $\sum_{i=0}^{n} abs(v_i)$

Table 5.1: Branch Transformation

| Predicates | Transformed Form | Remarks |
|---|---|---|
| $a == b$ | $(a + var1) \geq b \wedge$ $(b + var2) \geq a$ | |
| $a \neq b$ | $a \neq b$ | No need for transformation |
| $a < b$ | $a < (b + var)$ | |
| $a > b$ | $(a + var) > b$ | |
| $a \leq b$ | $a \leq (b + var)$ | |
| $a \geq b$ | $(a + var) \geq b$ | |
| $a \vee b$ | Apply transformation to expr $a$ and\or $b$ using the above rules | |
| $a \wedge b$ | Apply transformation to expr $a$ and\or $b$ using the above rules | |

in a 'controlled' way.

### 5.4.2 Transformation Rules

We define simple rules for adding additional variables to the existing variables of 'difficult' branches. Table 5.1 shows how assertions in a program may be stretched. Some times it is relatively difficult to add extra variables directly to the actual expression. For example in the case of "equality, ==" operator, we can not add a variable directly, as it will lead to erroneous data. In this case we also transform operator first into an equivalent form i.e., $(a \geq b \wedge b \geq a)$ and then add the additional variables.

A zero cost solution to an identified goal provides test data that achieves the goal for the original program. The extension of the cost function to include punishment of non-zero values of additional variables does not assume any particular mechanism for the traditional cost function component. We have chosen the traditional one with which we are most familiar, but others' approaches can be simply extended in the way we suggest.

## 5.5 Experiments and Evaluation

This section describes the experiments carried out to assess the program stretching technique. The aims of the experiments are to:

- Demonstrate the validity of the program stretching concept.
- Show that the technique can be applied both at code and architectural levels.

We performed two sets of experiments. In the first set, we considered three small case studies of program code of varied McCabe structural complexity. These programs are given in Appendix B. In the second set of experiments, we applied the concept at the architectural level to MATLAB® Simulink® models. We compared performance of the programs on the basis of (i) coverage, i.e., input test data found for branches against total branches to satisfy a coverage criterion, (ii) success rate, i.e., how many times test data was found when a program was run multiple times to satisfy a coverage criterion and, (iii) the average number of executions taken to find the test data.

### 5.5.1 Experiment Set 1: Code Examples

We used Simulated Annealing (SA) with the following parameters for our first set of experiments. Further experiments can be carried out to optimise the parameters. However, since we kept the same parameters for both approaches, we believe the results will be similar for the optimise set as well.

Move strategy: fixed

Geometric Temperature decrease rate: 0.8

Number of iteration in inner loop: 250

Maximum number of iteration in outer loop: 1000

Stopping criteria: Either a solution is found or the maximum number of iterations are reached.

Program 1 is relatively easy having a McCabe structural complexity of 10 and is experimented with to find the applicability of the stretching principle to different relational operators. Program 2 though has a structural complexity of 8 but is

relatively difficult from a search point of view because of the branching structure. Program 3 has a structural complexity of 14 and the branching structure is more difficult for search. In each of the above programs, our goal is to achieve the coverage of the last branch as indicated in the appendix. To reach that goal we 'stretched' the intermediate difficult branches.

### 5.5.2 Analysis and Evaluation of Experiment Set 1

With program 1, we were able to obtain 100% coverage and success rate using the traditional search based approach and also using program stretching. However, as expected, the average number of iterations to cover the required target in the stretched program approach was more than for the traditional approach. Although the goal is satisfied with the stretched program approach, the search process must expend additional effort reducing the program to the original and dragging the test data with it to maintain goal satisfaction. For easy goals, we believe that the traditional approach should be used instead of program stretching.

In program 2, the search was again 100% successful in each case. However, the task here is more difficult and the results are better for the program stretching approach. Program stretching, in this case, would appear to give efficiency advantages.

In program 3, the results were even better. In this case the search process was successful in 92% of runs for the standard approach and 99.8% of runs using program stretching. The average number of function evaluations (program executions) for successful runs were also better for the program streching approach. Tables 5.2 and 5.3 summarise the above results.

### 5.5.3 Experiment Set 2: Simulink Models

As stated above, for the second set of experiments we used MATLAB Simulink models. For this set of experiments we again used the models from Zhan [196]. These models have been described in Chapter 4 in detail. We used the following configuration for SA parameters. The parameters setting is based on the

Table 5.2: No of executions to generate test data Via Standard SBTDG (Based on 500 runs)

| | Program1 | Program2 | Program 3 |
|---|---|---|---|
| Average | 4825 | 14248 | 78212 |
| Max | 15037 | 24315 | 250050 |
| Min | 451 | 2663 | 26604 |
| SD | 2876 | 5564 | 58747 |
| Success Rate(%) | 100 | 100 | 92 |

Table 5.3: No of executions to generate test data via Program Stretching (Based on 500 runs)

| | Program1 | Program2 | Program 3 |
|---|---|---|---|
| Average | 7222 | 11976 | 61960 |
| Max | 24900 | 18405 | 140606 |
| Min | 1341 | 5514 | 14411 |
| SD | 3671 | 2195 | 35853 |
| Success Rate(%) | 100 | 100 | 99.8 |

experimental work by Zhan and Clark [197].

Move strategy: fixed.

Geometric Temperature decrease rate: 0.9

No of inner loop iterations: 100

Max. No. of outer loop iterations: 300

Stopping criteria: Either a solution is found or maximum number of iterations are reached.

### 5.5.4 Analysis and Evaluation of Experiment Set 2

We considered only those branches for analysis for which the traditional approach was not one hundred percent successful. Zhan and Clark [196] used four models. Model *SmplSw* is rather straightforward and hence is not considered for experiments. *Quadratic* model is also relatively simple but the search is made more difficult by introducing local minima. This was achieved by changing the range of input variables.

*Quadratic* model contains three Switch blocks and hence eight 'paths' or, more specifically combinations. *RandMdl* has four Switch blocks and hence has sixteen combinations. *Combine* model has seven Switch blocks and hence one hundred and twenty eight combinations. We targeted each of these combinations in the cases of Quadratic and RandMdl models. We 'stretched' the models for the case where all switch blocks take the value true. Each model was run thirty times for each of the combination then to achieve statistically significant results. For example, for RandMdl, we did four hundred and eighty runs in total.

The *Quadratic* model gave much better results for the stretching approach. However, for models *RandMdl* and *Combine* the results in both approaches are very similar in terms of success rate. But the 'stretching' is more expensive as it requires more number of executions to find the input data to cover the required combination. The results have been summarised in Tables 5.4 and 5.5.

In both sets of above experiments, the neighbourhood is searched by considering a small change in any of the variables, either actual or auxiliary, and then the modified fitness function is evaluated. All the variables have equal probability to be selected for the next move. The move is always accepted, if the fitness function value is improved. However, by restricting the neighbourhood search, we found significant improvement in the results. In such case, when the auxiliary variable is chosen for the next move and its next neighbouring value is chosen, one of the actual variable is also changed by a fraction of that value. All the actual variables have the same probability to be selected for this change. The fitness function due to all the actual variables, as well as the total fitness function are evaluated. The move is accepted if both fitness functions are improved or if there is an improvement in any of the fitness functions but the other remains unchanged. Table 5.6 summarises the results for Simulink models for this strategy. We can see that there is a clear improvement in the average number of executions as well as the success rate.

Tables 5.7 to 5.13 show the coverages and success rates for all the combinations in all three models.

Table 5.4: SBTDG for Simulink Models without Program Stretching (Based on 30 runs)

| | Quadratic | RandMdl | Combine |
|---|---|---|---|
| Total No of Branches | 8 | 16 | 116 |
| No of Branches for Analysis | 4 | 12 | 2 |
| Mean Success Rate(%) | 49.16 | 57.5 | 33 |
| Coverage(%) | 100 | 100 | 100 |
| Mean No of Executions | 587 | 2021 | 3122 |

Table 5.5: SBTDG for Simulink Models with Program Stretching (Based on 30 runs)

| | Quadratic | RandMdl | Combine |
|---|---|---|---|
| Success Rate(%) | 90.83 | 56.89 | 36.33 |
| Coverage(%) | 100 | 100 | 100 |
| Mean No of Executions | 5769 | 5857 | 7350 |

Table 5.6: SBTDG for Simulink Models with Program Stretching and (Stretch and Restrict) (Based on 30 runs)

| | Quadratic | RandMdl | Combine |
|---|---|---|---|
| Success Rate(%) | 90.83 | 80 | 77 |
| Coverage(%) | 100 | 100 | 100 |
| Mean No of Executions | 3885 | 7935 | 6534 |

Table 5.7: Quadratic Model without Program Stretching

| No | Combination | Unmodified | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 | 19 | 261 | 506 | 32 |
| 2 | 1 1 -1 | 16 | 1335 | 547 | 136 |
| 3 | 1 -1 1 | 15 | 387 | 790 | 90 |
| 4 | -1 1 1 | 30 | 103 | 369 | 1 |
| 5 | 1 -1 -1 | 09 | 363 | 512 | 37 |
| 6 | -1 -1 1 | 30 | 262 | 555 | 12 |
| 7 | -1 1 -1 | 30 | 84 | 319 | 1 |
| 8 | -1 -1 -1 | 30 | 238 | 480 | 1 |

Table 5.8: Quadratic Model with Program Stretching

| No | Combination | Program Stretching Random | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 | 28 | 6819 | 15739 | 2142 |
| 2 | 1 1 -1 | 26 | 5559 | 11029 | 1458 |
| 3 | 1 -1 1 | 28 | 5685 | 15505 | 2397 |
| 4 | -1 1 1 | 30 | 1694 | 5487 | 548 |
| 5 | 1 -1 -1 | 27 | 5015 | 8371 | 2076 |
| 6 | -1 -1 1 | 30 | 1804 | 4392 | 470 |
| 7 | -1 1 -1 | 30 | 1677 | 5840 | 546 |
| 8 | -1 -1 -1 | 30 | 1787 | 4213 | 606 |

Table 5.9: Quadratic Model with Program Stretching (Stretch and Restrict)

| No | Combination | Program Stretching Stretch and Restrict | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 | 29 | 3429 | 9190 | 1196 |
| 2 | 1 1 -1 | 25 | 4166 | 9682 | 1777 |
| 3 | 1 -1 1 | 30 | 3993 | 12823 | 1309 |
| 4 | -1 1 1 | 30 | 1994 | 8293 | 417 |
| 5 | 1 -1 -1 | 25 | 3955 | 8415 | 1568 |
| 6 | -1 -1 1 | 30 | 1894 | 6337 | 486 |
| 7 | -1 1 -1 | 30 | 1964 | 7599 | 533 |
| 8 | -1 -1 -1 | 30 | 2054 | 5465 | 623 |

Table 5.10: Random Model without Program Stretching

| NO | Combination | Unmodified | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 1 | 7 | 630 | 1153 | 1 |
| 2 | 1 1 1 -1 | 19 | 1417 | 2349 | 268 |
| 3 | 1 1 -1 1 | 15 | 280 | 1296 | 1 |
| 4 | 1 -1 1 1 | 12 | 755 | 1605 | 13 |
| 5 | -1 1 1 1 | 30 | 2396 | 4421 | 102 |
| 6 | 1 1 -1 -1 | 17 | 122 | 504 | 1 |
| 7 | 1 -1 -1 1 | 5 | 476 | 1200 | 1 |
| 8 | -1 -1 1 1 | 13 | 2993 | 7877 | 1295 |
| 9 | 1 -1 1 -1 | 5 | 3537 | 8715 | 1127 |
| 10 | -1 1 1 -1 | 30 | 3154 | 5156 | 1 |
| 11 | -1 1 -1 1 | 30 | 844 | 3542 | 1 |
| 12 | 1 -1 -1 -1 | 11 | 4680 | 5478 | 104 |
| 13 | -1 -1 -1 1 | 16 | 4748 | 8748 | 3534 |
| 14 | -1 -1 1 -1 | 21 | 4157 | 15131 | 1 |
| 15 | -1 1 -1 -1 | 30 | 1046 | 3441 | 1 |
| 16 | -1 -1 -1 -1 | 15 | 1101 | 4167 | 1 |

Table 5.11: Random Model with Program Stretching

| No | Combination | Program Stretching Random | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 1 | 13 | 5429 | 12856 | 1204 |
| 2 | 1 1 1 -1 | 13 | 6574 | 16670 | 1527 |
| 3 | 1 1 -1 1 | 08 | 2300 | 4438 | 881 |
| 4 | 1 -1 1 1 | 09 | 4262 | 9698 | 1611 |
| 5 | -1 1 1 1 | 30 | 6420 | 20011 | 1792 |
| 6 | 1 1 -1 -1 | 20 | 4806 | 11156 | 557 |
| 7 | 1 -1 -1 1 | 8 | 3498 | 5952 | 1306 |
| 8 | -1 -1 1 1 | 9 | 5154 | 8435 | 1304 |
| 9 | 1 -1 1 -1 | 10 | 10066 | 18472 | 3398 |
| 10 | -1 1 1 -1 | 30 | 7098 | 19292 | 1441 |
| 11 | -1 1 -1 1 | 30 | 4164 | 11215 | 1011 |
| 12 | 1 -1 -1 -1 | 10 | 7901 | 19790 | 5059 |
| 13 | -1 -1 -1 1 | 19 | 6603 | 14869 | 4446 |
| 14 | -1 -1 1 -1 | 13 | 6923 | 19247 | 1299 |
| 15 | -1 1 -1 -1 | 30 | 5774 | 13171 | 1411 |
| 16 | -1 -1 -1 -1 | 17 | 6752 | 19219 | 742 |

Table 5.12: Random Model with Program Stretching (Stretch and Restrict)

| No | Combination | Program Stretching Stretch and Restrict | | | |
|---|---|---|---|---|---|
| | | Success Rate | No of Executions | Max | Min |
| 1 | 1 1 1 1 | 30 | 7051 | 25972 | 1134 |
| 2 | 1 1 1 -1 | 29 | 9286 | 24735 | 818 |
| 3 | 1 1 -1 1 | 30 | 13187 | 25168 | 1676 |
| 4 | 1 -1 1 1 | 13 | 6143 | 19824 | 1040 |
| 5 | -1 1 1 1 | 30 | 6765 | 24895 | 1077 |
| 6 | 1 1 -1 -1 | 30 | 9056 | 26804 | 1136 |
| 7 | 1 -1 -1 1 | 21 | 7052 | 19942 | 988 |
| 8 | -1 -1 1 1 | 6 | 11452 | 21303 | 2222 |
| 9 | 1 -1 1 -1 | 14 | 6845 | 15259 | 2988 |
| 10 | -1 1 1 -1 | 30 | 6877 | 22632 | 999 |
| 11 | -1 1 -1 1 | 30 | 6816 | 19644 | 1306 |
| 12 | 1 -1 -1 -1 | 20 | 8387 | 15788 | 1841 |
| 13 | -1 -1 -1 1 | 24 | 9155 | 24656 | 5335 |
| 14 | -1 -1 1 -1 | 20 | 7914 | 13366 | 583 |
| 15 | -1 1 -1 -1 | 30 | 5553 | 29966 | 888 |
| 16 | -1 -1 -1 -1 | 27 | 5426 | 13530 | 756 |

Table 5.13: Combine Model with Program Stretching (Stretch and Restrict)

| No | Combination | Success Rate | No of Executions | Max | Min |
|---|---|---|---|---|---|
| 1 | 1 1 1 1 1 1 1 | 23 | 6533 | 9403 | 4001 |
| 2 | 1 1 1 1 1 1 -1 | 6 | 8940 | 16737 | 3626 |
| 3 | 1 1 1 1 1 -1 1 | 11 | 5058 | 12148 | 2556 |
| 4 | 1 1 1 1 1 -1 -1 | 4 | 10024 | 14674 | 4576 |
| 5 | 1 1 1 1 -1 1 1 | 10 | 6022 | 12349 | 2142 |
| 6 | 1 1 1 1 -1 1 -1 | 10 | 6508 | 13633 | 3061 |
| 7 | 1 1 1 1 -1 -1 1 | 22 | 6973 | 20461 | 2939 |
| 8 | 1 1 1 1 -1 -1 -1 | 30 | 8691 | 24721 | 2496 |
| 9 | 1 1 1 -1 1 1 1 | 13 | 10639 | 27067 | 3092 |
| 10 | 1 1 1 -1 1 1 -1 | 12 | 18235 | 21674 | 3228 |
| 11 | 1 1 1 -1 1 -1 1 | 28 | 6814 | 25033 | 2228 |
| 12 | 1 1 1 -1 1 -1 -1 | 30 | 7131 | 19498 | 2623 |
| 13 | 1 1 1 -1 -1 1 1 | 7 | 5800 | 8369 | 3019 |
| 14 | 1 1 1 -1 -1 1 -1 | 17 | 4342 | 6271 | 3126 |
| 15 | 1 1 1 -1 -1 -1 1 | 30 | 4525 | 10422 | 2210 |
| 16 | 1 1 1 -1 -1 -1 -1 | 30 | 5263 | 17487 | 1941 |
| 17 | 1 1 -1 1 1 1 1 | 6 | 9220 | 12320 | 6089 |
| 18 | 1 1 -1 1 1 1 -1 | 6 | 9715 | 16407 | 4057 |
| 19 | 1 1 -1 1 1 -1 1 | 8 | 7015 | 4590 | 3058 |
| 20 | 1 1 -1 1 1 -1 -1 | 2 | 10225 | 13456 | 7003 |
| 21 | 1 1 -1 1 -1 1 1 | 17 | 6536 | 11063 | 3123 |
| 22 | 1 1 -1 1 -1 1 -1 | 10 | 7141 | 8597 | 2188 |
| 23 | 1 1 -1 1 -1 -1 1 | 21 | 7750 | 21385 | 2567 |
| 24 | 1 1 -1 1 -1 -1 -1 | 30 | 8112 | 20172 | 2337 |
| 25 | 1 1 -1 -1 1 1 1 | 15 | 10738 | 22885 | 2678 |

Table 5.13: Combine Model with Program Stretching (Stretch and Restrict)

| No | Combination | Success Rate | No of Executions | Max | Min |
|---|---|---|---|---|---|
| 26 | 1 1 -1 -1 1 1 -1 | 10 | 9454 | 23954 | 2530 |
| 27 | 1 1 -1 -1 1 -1 1 | 15 | 8094 | 16639 | 3552 |
| 28 | 1 1 -1 -1 1 -1 -1 | 19 | 10717 | 17381 | 3266 |
| 29 | 1 1 -1 -1 -1 1 1 | 18 | 5435 | 11663 | 2730 |
| 30 | 1 1 -1 -1 -1 1 -1 | 15 | 4206 | 8209 | 1985 |
| 31 | 1 1 -1 -1 -1 -1 1 | 30 | 6498 | 19626 | 2526 |
| 32 | 1 1 -1 -1 -1 -1 -1 | 30 | 6221 | 120769 | 1914 |
| 33 | 1 -1 1 1 1 1 1 | 9 | 6747 | 4500 | 3999 |
| 34 | 1 -1 1 1 1 1 -1 | 4 | 11088 | 21522 | 3797 |
| 35 | 1 -1 1 1 1 -1 1 | 11 | 7155 | 13231 | 2174 |
| 36 | 1 -1 1 1 1 -1 -1 | 4 | 5103 | 26767 | 3488 |
| 37 | 1 -1 1 1 -1 1 1 | 14 | 6479 | 12704 | 2400 |
| 38 | 1 -1 1 1 -1 1 -1 | 12 | 7435 | 9516 | 3156 |
| 39 | 1 -1 1 1 -1 -1 1 | 21 | 4923 | 12888 | 2834 |
| 40 | 1 -1 1 1 -1 -1 -1 | 30 | 6136 | 15256 | 2905 |
| 41 | 1 -1 1 -1 1 1 1 | 16 | 8521 | 19626 | 3151 |
| 42 | 1 -1 1 -1 1 1 -1 | 16 | 11239 | 27638 | 2924 |
| 43 | 1 -1 1 -1 1 -1 1 | 18 | 5822 | 7442 | 2730 |
| 44 | 1 -1 1 -1 1 -1 -1 | 15 | 4164 | 7424 | 2358 |
| 45 | 1 -1 1 -1 -1 1 1 | 15 | 5406 | 11995 | 2608 |
| 46 | 1 -1 1 -1 -1 1 -1 | 16 | 4169 | 9706 | 2 |
| 47 | 1 -1 1 -1 -1 -1 1 | 30 | 6257 | 19695 | 2521 |
| 48 | 1 -1 1 -1 -1 -1 -1 | 30 | 4865 | 11653 | 2252 |
| 49 | 1 -1 -1 1 1 1 1 | 7 | 8847 | 16873 | 3042 |
| 50 | 1 -1 -1 1 1 1 -1 | 3 | 11890 | 15150 | 8786 |
| 51 | 1 -1 -1 1 1 -1 1 | 7 | 6241 | 9679 | 3101 |

Table 5.13: Combine Model with Program Stretching (Stretch and Restrict)

| No | Combination | Success Rate | No of Executions | Max | Min |
|---|---|---|---|---|---|
| 52 | 1 -1 -1 1 1 -1 -1 | 4 | 7735 | 12001 | 5249 |
| 53 | 1 -1 -1 1 -1 1 1 | 18 | 7630 | 17050 | 2623 |
| 54 | 1 -1 -1 1 -1 1 -1 | 14 | 8691 | 13503 | 2552 |
| 55 | 1 -1 -1 1 -1 -1 1 | 19 | 9913 | 22630 | 2894 |
| 56 | 1 -1 -1 1 -1 -1 -1 | 30 | 10454 | 27328 | 2708 |
| 57 | 1 -1 -1 -1 1 1 1 | 17 | 12254 | 22972 | 2803 |
| 58 | 1 -1 -1 -1 1 1 -1 | 16 | 12042 | 26684 | 3053 |
| 59 | 1 -1 -1 -1 1 -1 1 | 8 | 6258 | 11514 | 3222 |
| 60 | 1 -1 -1 -1 1 -1 -1 | 8 | 7574 | 13471 | 3600 |
| 61 | 1 -1 -1 -1 -1 1 1 | 20 | 6123 | 8609 | 2195 |
| 62 | 1 -1 -1 -1 -1 1 -1 | 15 | 5788 | 8111 | 2275 |
| 63 | 1 -1 -1 -1 -1 -1 1 | 30 | 5993 | 18262 | 2625 |
| 64 | 1 -1 -1 -1 -1 -1 -1 | 30 | 5224 | 16319 | 2652 |
| 65 | -1 1 1 1 1 1 1 | 7 | 6674 | 14842 | 3390 |
| 66 | -1 1 1 1 1 1 -1 | 8 | 8290 | 13954 | 5880 |
| 67 | -1 1 1 1 1 -1 1 | 9 | 6090 | 11807 | 3055 |
| 68 | -1 1 1 1 1 -1 -1 | 7 | 6237 | 10920 | 3329 |
| 69 | -1 1 1 1 -1 1 1 | 16 | 6745 | 12738 | 3838 |
| 70 | -1 1 1 1 -1 1 -1 | 14 | 5622 | 10492 | 2634 |
| 71 | -1 1 1 1 -1 -1 1 | 19 | 5971 | 9756 | 2815 |
| 72 | -1 1 1 1 -1 -1 -1 | 30 | 7541 | 16640 | 3302 |
| 73 | -1 1 1 -1 1 1 1 | 20 | 10578 | 26237 | 4635 |
| 74 | -1 1 1 -1 1 1 -1 | 16 | 7510 | 20426 | 3716 |
| 75 | -1 1 1 -1 1 -1 1 | 11 | 7425 | 13885 | 3239 |
| 76 | -1 1 1 -1 1 -1 -1 | 16 | 5894 | 8669 | 3583 |
| 77 | -1 1 1 -1 -1 1 1 | 15 | 5878 | 9947 | 3152 |

Table 5.13: Combine Model with Program Stretching (Stretch and Restrict)

| No | Combination | Success Rate | No of Executions | Max | Min |
|---|---|---|---|---|---|
| 78 | -1 1 1 -1 -1 1 -1 | 17 | 4960 | 12687 | 3010 |
| 79 | -1 1 1 -1 -1 -1 1 | 30 | 7856 | 19382 | 2392 |
| 80 | -1 1 1 -1 -1 -1 -1 | 30 | 7045 | 20827 | 2137 |
| 81 | -1 1 -1 1 1 1 1 | 8 | 10274 | 18158 | 4841 |
| 82 | -1 1 -1 1 1 1 -1 | 5 | 11356 | 20010 | 4645 |
| 83 | -1 1 -1 1 1 -1 1 | 12 | 7235 | 12367 | 4108 |
| 84 | -1 1 -1 1 1 -1 -1 | 7 | 11509 | 15277 | 6509 |
| 85 | -1 1 -1 1 -1 1 1 | 18 | 9347 | 15701 | 3747 |
| 86 | -1 1 -1 1 -1 1 -1 | 9 | 7590 | 14278 | 2846 |
| 87 | -1 1 -1 1 -1 -1 1 | 19 | 9354 | 17189 | 4523 |
| 88 | -1 1 -1 1 -1 -1 -1 | 29 | 8656 | 24518 | 2647 |
| 89 | -1 1 -1 -1 1 1 1 | 15 | 12454 | 27337 | 4693 |
| 90 | -1 1 -1 -1 1 1 -1 | 20 | 8371 | 26667 | 2 |
| 91 | -1 1 -1 -1 1 -1 1 | 13 | 11170 | 26466 | 3253 |
| 92 | -1 1 -1 -1 1 -1 -1 | 14 | 7201 | 13951 | 3924 |
| 93 | -1 1 -1 -1 -1 1 1 | 12 | 6140 | 8617 | 4119 |
| 94 | -1 1 -1 -1 -1 1 -1 | 14 | 5959 | 13532 | 3551 |
| 95 | -1 1 -1 -1 -1 -1 1 | 30 | 5870 | 20660 | 2912 |
| 96 | -1 1 -1 -1 -1 -1 -1 | 30 | 5900 | 18147 | 2701 |
| 97 | -1 -1 1 1 1 1 1 | 8 | 5331 | 9307 | 3468 |
| 98 | -1 -1 1 1 1 1 -1 | 6 | 11285 | 16582 | 5377 |
| 99 | -1 -1 1 1 1 -1 1 | 15 | 5250 | 9665 | 2717 |
| 100 | -1 -1 1 1 1 -1 -1 | 3 | 7690 | 11213 | 5185 |
| 101 | -1 -1 1 1 -1 1 1 | 16 | 7139 | 12687 | 3514 |
| 102 | -1 -1 1 1 -1 1 -1 | 13 | 5719 | 8278 | 3954 |
| 103 | -1 -1 1 1 -1 -1 1 | 17 | 8020 | 15286 | 3244 |

Table 5.13: Combine Model with Program Stretching (Stretch and Restrict)

| No | Combination | Success Rate | No of Executions | Max | Min |
|---|---|---|---|---|---|
| 104 | -1 -1 1 1 -1 -1 -1 | 30 | 7963 | 25128 | 2162 |
| 105 | -1 -1 1 -1 1 1 1 | 14 | 7701 | 20337 | 2970 |
| 106 | -1 -1 1 -1 1 1 -1 | 19 | 14551 | 24092 | 3511 |
| 107 | -1 -1 1 -1 1 -1 1 | 13 | 5752 | 14027 | 3502 |
| 108 | -1 -1 1 -1 1 -1 -1 | 9 | 6520 | 11976 | 3747 |
| 109 | -1 -1 1 -1 -1 1 1 | 13 | 6220 | 11863 | 3228 |
| 110 | -1 -1 1 -1 -1 1 -1 | 19 | 5200 | 10701 | 3190 |
| 111 | -1 -1 1 -1 -1 -1 1 | 30 | 6775 | 23559 | 2903 |
| 112 | -1 -1 1 -1 -1 -1 -1 | 30 | 7222 | 22084 | 2953 |
| 113 | -1 -1 -1 1 1 1 1 | 3 | 7990 | 14812 | 4579 |
| 114 | -1 -1 -1 1 1 1 -1 | 5 | 10564 | 11805 | 8221 |
| 115 | -1 -1 -1 1 1 -1 1 | 8 | 9550 | 18097 | 4328 |
| 116 | -1 -1 -1 1 1 -1 -1 | 2 | 9130 | 13367 | 4894 |
| 117 | -1 -1 -1 1 -1 1 1 | 16 | 7788 | 15660 | 4496 |
| 118 | -1 -1 -1 1 -1 1 -1 | 13 | 6181 | 13041 | 3067 |
| 119 | -1 -1 -1 1 -1 -1 1 | 18 | 9613 | 24955 | 4104 |
| 120 | -1 -1 -1 1 -1 -1 -1 | 30 | 9956 | 24552 | 3087 |
| 121 | -1 -1 -1 -1 1 1 1 | 11 | 8895 | 27839 | 4133 |
| 122 | -1 -1 -1 -1 1 1 -1 | 12 | 13485 | 29321 | 5481 |
| 123 | -1 -1 -1 -1 1 -1 1 | 14 | 6923 | 13212 | 3256 |
| 124 | -1 -1 -1 -1 1 -1 -1 | 14 | 7497 | 19504 | 3928 |
| 125 | -1 -1 -1 -1 -1 1 1 | 16 | 6049 | 9574 | 3710 |
| 126 | -1 -1 -1 -1 -1 1 -1 | 11 | 5045 | 8792 | 2839 |
| 127 | -1 -1 -1 -1 -1 -1 1 | 30 | 6388 | 18753 | 2816 |
| 128 | -1 -1 -1 -1 -1 -1 -1 | 30 | 7279 | 24313 | 2561 |

## 5.6 Conclusions

The program stretching principle is in its initial stages. Initial results are promising and we believe that the technique can be extended to more complex systems. The technique was motivated by the need to satisfy "difficult" goals. The studies indeed show that program stretching has advantages in these cases, either showing greater success or greater efficiency (or both) than standard search based test data generation.

The approach is basically a form of continuous program transformation. A key difference here is that the transformations do not preserve program semantics. The idea of stretching was originally motivated as a form of "topological" deformation. Ideally the input domains corresponding to satisfying test goals in the stretched program would map in a straightforward way to those of the original program's test goals. This is not essential. One could easily imagine some input partitions vanishing as the program is stretched — some paths through the program may no longer be possible for example.

In this chapter we have shown how program stretching can be used to find hard-to-find branches, but really this is about generating test data to satisfy hard-to-find conditions.

## 5.7 Application Outside SBSE

We believe that program stretching is a novel yet appealing concept. Although developed to solve a specific problem in SBTDG, it seems plausible that the approach could find application elsewhere. In abstract terms the fundamental idea is that the "problem" and "solution" are developed together. The problem is essentially relaxed (made easier) until a satisficing solution is found. An attempt is then made to migrate the problem to the original problem of interest with the solution being "dragged" along to maintain satisfaction of the changing constraints. We see no reason in principle why this form of relaxing and dragging should not find wider application within the search based engineering disciplines.

Chapter 6

# Strengthening Inferred Specifications using Search Based Testing

## 6.1 Introduction

The software specification captures the required behaviour of a program. It is an important 'document' used throughout the software development lifecycle. Specifications are informal plain text statements of needs, semi-formal structured or graphical descriptions, or statements with mathematical precision (usually referred to as formal specifications). Specifications may be identifiable documents in their own right, or else comprise assertion fragments embedded within code (e.g. as in the Design By Contract paradigm).

The extent to which a specification can be useful depends upon its specific form — each format has its own strengths and weaknesses. Formal specifications typically facilitate the automation of a variety of tasks (e.g. test data generation or proofs of correctness) but generally require a high level of skill to produce and read. Informal specifications are usable by a wider audience but may suffer from ambiguity.

In many cases we do not have any explicit specification. This is highly undesirable; specifications are highly useful documents to many stakeholders. Since

generating and maintaining specifications is a tedious job, it can be greatly beneficial if the process is automated. Work has been done in this respect, exploring the use of static code analysis techniques (e.g. [39, 168]) and dynamic techniques (most prominently the work of Ernst [59, 58, 60])

An invariant is a property of a program which remains true for all its executions (in case of a loop, it remains true in the beginning and at the end of each iteration of the loop) and hence represents a partial specification. There are unlimited number of program invariants. Some will be fundamental (the set of invariants defining the program behaviour) and others may be derived as consequences.

Static analysis techniques for deriving invariants from program code are sound theoretically, but in practice they are difficult to implement. One recent approach used to overcome such problems is dynamic (runtime) analysis [58]. In this approach likely invariants are inferred from the actual execution traces of the program when exercised by test cases. Since the inferred invariants are largely dependent on test cases, they may not be correct. To get around this problem, many approaches have been proposed [138, 192, 72, 50, 148].

In this chapter we present the use of search based test data generation techniques to verify and 'strengthen' the inferred putative invariants. An attempt is made to falsify inferred invariants using the available search techniques. The result is that an inferred specification can be iteratively challenged by search based test data generation techniques resulting in a more credible set of invariants.

## 6.2 Background

### 6.2.1 Dynamic Invariant Generation

Dynamic Invariant Generation techniques have in recent years attracted significant research interest. As stated above, data is collected from the state and IO execution traces of programs when test cases are run. Assertions can be generated over variables of interest. Assertions that are *consistent* with all traces are *possible* invariants. There are, of course, an infinity of such true statements but Ernst et al. [58, 152] have demonstrated how *useful* invariants can be generated. In particular

they have shown how invariants that make up the specification of a program can be generated. This "specification" is a best effort attempt to capture abstractly the behaviour of the implemented program. Daikon [60], their tool framework, uses a multi-step approach to inferring likely invariants. An instrumented version of a program (with code to record state data at program points during execution) is exercised with test cases. The likely invariants are detected and can be reported in many useful formats. The output consists of procedure pre- and post-conditions and generalized object invariants. Daikon checks for 75 different types of invariant and the extension mechanism is simple enough to include more. It also checks for conditional invariants and implications. A conditional invariant is only true part of the time. Consider the post condition for the absolute value procedure:

$if\ (arg < 0)$
$\quad then$ return== $-arg$
$\quad else$ return== $arg$

This is an example of conditional invariants [8]. Support for many popular programming languages has already been provided and can be further extended easily to other languages. In this thesis work we have considered the likely generated invariants at public methods' entry and exit.

### 6.2.2 SBTDG for Invariant Falsification

SBTDG techniques find input data that cause identified assertions to be true or false (as required). For specification strengthening purposes we simply target those assertions generated by dynamic techniques such as Daikon as fragments of inferred specifications. We need only represent such assertions in a form that is amenable to search. This can be readily obtained by representing an invariant in branch predicate form. Suppose we have a proposed invariant *inv* for some identified point in the program. If we insert a program statement "if(!inv);" at that point we can view the falsification of the invariant as a branch reachability

problem [70]. This enables us to use the standard SBTDG techniques, where we try to find test data to satisfy this branch. The overall approach we adopt can readily be used with any reasonably effective SBTDG tool.

## 6.3 Related Work

Nimmer et al. [138] proposed a combination of dynamic and static analysis techniques to generate specifications and prove their correctness. They used Daikon with ESC/Java [51, 105]. Their work shows that specifications generated from program execution are reasonably accurate. However, due to limitations of the tools spurious inferred invariants may remain.

Harder et al. [75] proposed the Operational Difference (OD) technique for generating, augmenting, and minimizing the test suites. The main idea is to generate an *operational abstraction* (OA), an abstraction of the program's runtime behaviour, from program executions and then try to improve it. The technique starts with an empty test suite and empty OA. Test cases are generated and evaluated by means of the change that it brings in the OA. A test case that improves the OA can be added to a test suite and a test case that doesn't can be removed. They also proposed *operational coverage* as a measure of difference between the OA and the correct specification. This is a relative term and requires the presence of an oracle to be computed. The technique developed fault revealing test suites, however, it is not guaranteed that the change brought by a test case in the OA is correct. For example, consider a variable $A >= 0$. A test case, say $A = 5$, may cause the OA to include $A\ != 0$ unless and until another test case $A = 0$ is executed. Thus elements of the OA may be untrue. Our technique on the other hand searches for such "missing" or revealing test cases and hence increases the quality of the OA.

Gupta [70] proposed modelling the invariant detection problem as a test data generation problem in a manner very similar to our approach and then suggested applying test data generation techniques to falsify the generated invariants. However they did not apply any technique in practice to assess it. They [72]

further proposed an invariant coverage criterion based on a set of definition-use chains of variables of an invariant property. The approach though increases the quality of likely generated invariants may generate many infeasible definition-use chains. Filtering out such infeasible chains is a tedious process. Also the test set generated to satisfy invariant coverage criterion may not be sufficient in detecting all the spurious likely invariants.

Hangal [74] and Xie [192] used specification violation approaches to improve their inferred specifications. Hangal's work [74] was mainly aimed at detecting bugs. It is implemented in the DIDUCE tool, which continually checks the program's behaviour against the invariants during program execution and reports all detected violations at the end. Xie's [192] approach uses Daikon with ParaSoft JTest [3] with the intention of improving the test suites for unit testing of Java programs.

Pacheco [148] proposed a technique that selects from a large test set, a small subset of test inputs that are likely to reveal faults in the software under test. Their technique infers an operational model of the software's operation from the correct execution of a program using the Daikon invariants detector. The program is then executed using randomly generated 'candidate' inputs with provided inferred invariants monitored to see if they are satisfied. A classifier system labels candidate inputs as *illegal, normal operation* or *fault-revealing*. The interesting ones are the fault-revealing inputs which may indicate a fault in the program. If more than one input violates the same property then only one input is selected from that set. The technique has been implemented in a tool called as *Eclat*. The technique was tested with some sample programs and compared with another tool, JCrasher [40]. Eclat yields good results. The technique is effective and revealed previously unknown faults. However, it requires a priori correct test suites which may not be available. It also sometimes classifies an input as fault-revealing, though it may not be so.

## 6.4 Model for Invariant Falsification

One limitation of many dynamic invariant inference approaches is that they are at the mercy of the test data. Test suites may not be wholly appropriate for the purposes of inference. We may also get spurious or 'not interesting' [72] invariants. Though some approaches [72, 75, 192] have been proposed to modify test suites, they do not completely preclude the generation of spurious invariants. This is especially the case when the domain of input data is large or the program is complex. We may have a test suite satisfying a certain structural criterion, but due to not enough test cases, we may get erroneous specifications. Figure 6.1 is an example of such a program. In order to refine the specification for such programs we adopt a systematic approach.

```
public class Prog{
        private int b = 10;
        private int c = 30;
        public void test(int a){
                b = 5 * a;
                c = 100000 – 5 * a;
        }
}
```

Figure 6.1: A simple program with a large input space

Our approach is similar to that of [72, 74, 192]. However, we propose a search based test data generation approach and unlike previous approaches, our domain of application is more 'procedural' in nature.

### 6.4.1 High Level Model

Figure 6.2 shows a high level model of our approach. We can divide our model into three main steps.

*Invariant Inference:* Invariants are generated from a randomly generated test suite containing 'enough' test cases to allow a reasonably succinct set of invariants to be inferred.

*Invariants' Class:* The invariants generated are then 'imported' into an intermediate class. It contains all the invariants in a format suitable to apply our

search based test data generation techniques.

*Invariant Falsification:* A test data generation tool attempts to falsify the invariants.

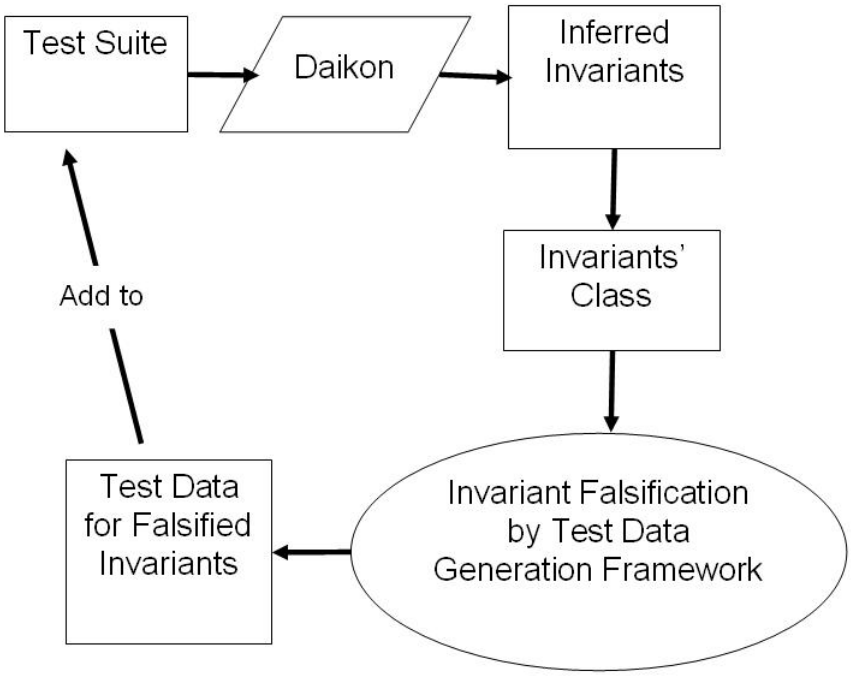


Figure 6.2: High Level Model

These are further elaborated below.

### 6.4.2 Invariant Inference

In this step we use Daikon to infer likely invariants. To begin with, a test suite of 'reasonably' large set of test cases is given as input to Daikon. Daikon dynamically infers a list of invariants based on the execution of test cases. The test suite is either generated randomly or by the test data generation tool, described in Chapter 3. We can ensure that identified structural criteria are satisfied, whichever approach is adopted. Though the inferred invariants are affected by the number of test cases in a test suite, there is no direct correlation between the size of test suite and the invariant inference process [75]. Therefore, we chose a test suite of arbitrary size, ensuring at the same time to include enough test cases that may satisfy the structural criteria many times. The reason for this is to get

a list of generalized likely invariants and to eliminate the less interesting ones e.g. invariants peculiar to a specific test set used, such as "x is one of $\{4, 8, 9\}$".

Consider the running example in Figure 6.1. We used Daikon to infer invariants using 15000 randomly chosen values of "$a$" between -1000 and 100000. The output from Daikon is shown in Figure 6.3. The invariant $a! = 0$ is not necessarily true, but since the probability of randomly choosing $a = 0$ as an input is 1/101001, we can see why a random process is unlikely to generate such a case (even with 15000 trials). 15000 trials would generate at most 15000 distinct values of "$a$" and the domain of "$a$" comprises 101001 of which 0 is but one element. Thus in random test data generation techniques, which seem to be the choice for invariants falsification in previous approaches [74, 192], we are very likely to get the spurious invariants. Note that stronger coverage criteria, such as *Invariant Coverage* are also not effective here as a single test case, e.g. $a = 2$, may give us the coverage for all these criteria.

Compared to other techniques, search based techniques directly target an inferred likely invariant and seek data to falsify it. This eliminates the need for deriving many test suites [75] or more complex criteria [72]. Search based test data generation techniques provide one approach to finding counter-examples. Our implementation approach transforms the generation of a counter-example into a problem of executing the true branch of $if(!invariant)$;. The advantage and disadvantages of using search based test data generation approaches are therefore exactly those of using such approaches to satisfy path (and other) criteria. As indicated in Section 2.1, search based approaches are not inhibited by the use of "difficult" language constructs (pointers, dynamic data structures, etc.).

### 6.4.3 Intermediate Invariants' Class

For counter-example generation purposes we insert additional fragments into the program. At point $P$ in the program if an invariant such as $(a! = x)$ is proposed then we insert a code fragment $if\ (a == x)$. In test data generation terms breaking the invariant corresponds to reaching the true branch of the inserted fragment. Thus we model breaking invariants as path satisfaction problems.

Fragments are inserted for each invariant we want to falsify.

### 6.4.4 Test Data Generation Framework

The generated class is then given as an input to the test data generation framework, which searches data to falsify the invariants.

If an invariant can not be falsified even after exhausting the search criteria, the search is terminated for that invariant and the invariant is assumed correct. Here we use the tool's abilities to target branch conditions. We terminate the search process when either a solution is found or a maximum numbers of trials have been made. If a test case is found that falsifies the invariant, it is added to the existing test suite. The process continues until test data to falsify all the invariants have been searched for.

The modified test suite is again executed by Daikon and a modified list of invariants is generated. The process is repeated and if no further invariant is falsified in the following iteration, the process is terminated. For the example program, all the falsified invariants were found in the first iteration thus giving us more refined specifications as shown in figure 6.4.

## 6.5 Experiments

This section describes the experiments carried out to assess the above mentioned technique. The aim of the experiments is to demonstrate that search based test data generation techniques can be applied effectively to refine sets of automatically generated likely invariants produced by the Daikon tool.

We performed experiments with five programs including the example program described earlier. Two programs i.e., Middle and WrapRoundCounter are taken from [174]. These programs were used for specification conformance and were originally written in Ada. CalDate, is taken from [115] and BubbleSort is a modified form of a program given in [47]. These programs have been given in the appendix. We have considered the invariants generated for the public methods. Details of the input domains, test cases, number of Invariants generated and

```
Prog:::OBJECT
this has only one value
this.b != 0
this.c != 0
this.b != this.c
==========
Prog.Prog():::EXIT
this.b == 10
this.c == 30
==========
Prog.test(int):::ENTER
this.b == 0  (mod 5)
this.c == 0  (mod 5)
a != 0
this.b != a
this.c != a
==========
Prog.test(int):::EXIT
this.b == 0  (mod 5)
this.c == 0  (mod 5)
this.b + this.c - 100000 == 0
this.b != orig(this.b)
this.b != orig(this.c)
this.b != orig(a)
this.b - 5 * orig(a) == 0
this.c != orig(this.b)
this.c != orig(this.c)
this.c != orig(a)
this.c + 5 * orig(a) - 100000 == 0
==========
```

Figure 6.3: Initial Output from Daikon.

numbers of falsified invariants are shown in the Table 6.1.

### 6.5.1 Middle Program

Middle program takes three input variables and returns a variable having a value between the other two. If two variables have same value then the third one is reported as the middle value.

```
Prog:::OBJECT
this.c != 0
this.b != this.c
==============================
Prog.Prog():::EXIT
this.b == 10
this.c == 30
==============================
Prog.test(int):::ENTER
this.b == 0  (mod 5)
this.c == 0  (mod 5)
==============================
Prog.test(int):::EXIT
this.b == 0  (mod 5)
this.c == 0  (mod 5)
this.b + this.c - 100000 == 0
(orig(a) == 0) ==> (this.b == 0)
(this.b == 0) ==> (orig(a) == 0)
this.b - 5 * orig(a) == 0
this.c != orig(a)
this.c + 5 * orig(a) - 100000 == 0
==============================
```

Figure 6.4: Final Output from Daikon.

### 6.5.2 WrapRoundCounter

WrapRoundCounter is a simple program which counts from 0 to 10 and then wrap-around back to 0 again. This program takes one input of type integer. If the input is greater than 10, then it returns 0. Otherwise it increments the input variable by 1. This program is included to investigate conditional invariants as well.

### 6.5.3 BubbleSort

This program takes an array of int values. If the array is not sorted, it sorts it elements in an ascending order.

Table 6.1: Case Studies

| Program | Input Domain | # of Test cases | Total # of Invariants | # of Falsified Invariants |
|---|---|---|---|---|
| Middle | -10 — 10 | 15000 | 19 | 1 |
| Middle | -100 — 100 | 15000 | 21 | 0 |
| Middle | -500 — 500 | 15000 | 22 | 1 |
| Middle | -1000 — 1000 | 15000 | 22 | 2 |
| Middle | -5000 — 5000 | 15000 | 25 | 2 |
| BubbleSort | -100 — 100 | 15000 | 4 | 0 |
| BubbleSort | -1000 — 1000 | 15000 | 4 | 0 |
| WrapRound-Counter | 0 — 11 | 100 | 8 | 0 |
| WrapRound-Counter | 0 — 100 | 100 | 8 | 0 |
| WrapRound-Counter (conditional) | 0 — 11 | 100 | 8 | 0 |
| Example-Program | -1000 — 100000 | 15000 | 16 | 8 |
| CalDate | various | 500 | 5 | 3 |
| CalDate | various | 1000 | 5 | 3 |
| CalDate | various | 15000 | 3 | 1 |

### 6.5.4 CalDate

A description of this program has been given in Chapter 3. It takes three input variables, *day, week and year* of type int. *day* has a domain of 0-30, *week* 0-6 and year can take any integer value between -4713 to 3000.

### 6.5.5 Analysis

In the case of *Middle* program, we experimented with different domains. The program was tested with fifteen thousand randomly generated test cases. Usually, if we keep the input domain small, we don't expect spurious invariants to be generated. However, in the case of *Middle* even though we kept the input domain very small, a spurious invariant was nevertheless inferred. By further increasing the input space, Daikon inferred a set containing no spurious invariant. However, when the domain of input values was further increased, additional spurious inva-

riants were inferred by Daikon. In each case the search based test data generation framework successfully generated test data to falsify these spurious invariants.

In the case of *BubbleSort*, we considered two different domains and fifteen thousand randomly generated tests. This is an example of program where the input domain does not have an effect on the inferred invariants. Daikon inferred a correct set of invariants in this case.

In the case of *WrapRoundCounter*, we again considered two different domains of input variables. A hundred test cases were generated randomly. In all cases eight likely invariants were inferred. No inferred invariant was falsified by the search tool. Upon inspection we found that in this case too Daikon inferred a correct set of invariants. Also there was no effect of changing the input domain because of the nature of program.

For the *example program*, shown earlier, we generated fifteen thousand tests. The initial set of inferred invariants contained sixteen invariants out of which eight were spurious. All were falsified by the search based test data generation framework.

In the case of *calDate* we changed the number of test cases rather than the domain as the domain needs to remain fixed. The search tool was able to eliminate the spurious inferred invariants in each case. However, as expected, the number of spurious invariants decreases with the test suite size. Again the final sets of invariants were identical in each case.

In all the above cases the initial test suites were generated randomly. However, we included sufficient test cases in each case to enable Daikon to infer a compatible set of likely invariants. It is possible that the number of inferred invariants may be different for another suite of test cases.

## 6.6 Conclusion

Specifications are important. In many cases, they will not exist or (often even worse) be out of date. Specification synthesis tools such as Daikon offer a promising solution to this problem. However, Daikon, and indeed other dynamic

inference tools, make inferences based on the traces of the program when executed with a given test set. Inferred specifications may differ between test sets used; this presents a problem. Coverage criteria may be postulated but there would appear to be no clear candidate. In many cases users resort to random testing whilst in others structural criteria are used. These often allow erroneous invariants to be inferred. Our work shows that extant search based software test data generation approaches can be used to stress each inferred invariant with a view to falsifying it. Test data that falsifies inferred invariants can be added to the test suite and the inference tool can be rerun. This leads to more accurate inferred specifications. Currently the search for test data is carried out automatically by our tool. However, instrumenting the program with the Daikon invariants to be falsified is at present a manual process though this can also be automated.

The approach shows that search based test data generation and specification inference are complementary. (Search based test data generation techniques can also be used to generate the initial test suite itself.) Our approach is a pragmatic way to enable the important task of specification inference to be improved.

Chapter 7

# Evaluation, Future Work and Conclusion

In this chapter we will evaluate and conclude the work. Future directions are also provided for further work.

## 7.1 Evaluation

The work in this dissertation provides support for the following hypothesis:

> ***Search based test data generation techniques can be extended to satisfy criteria at both code and higher levels with increasing sophistication.***

To address the hypothesis four topics were considered. Figure 7.1 summarises these topics. In the following section we evaluate these in turn.

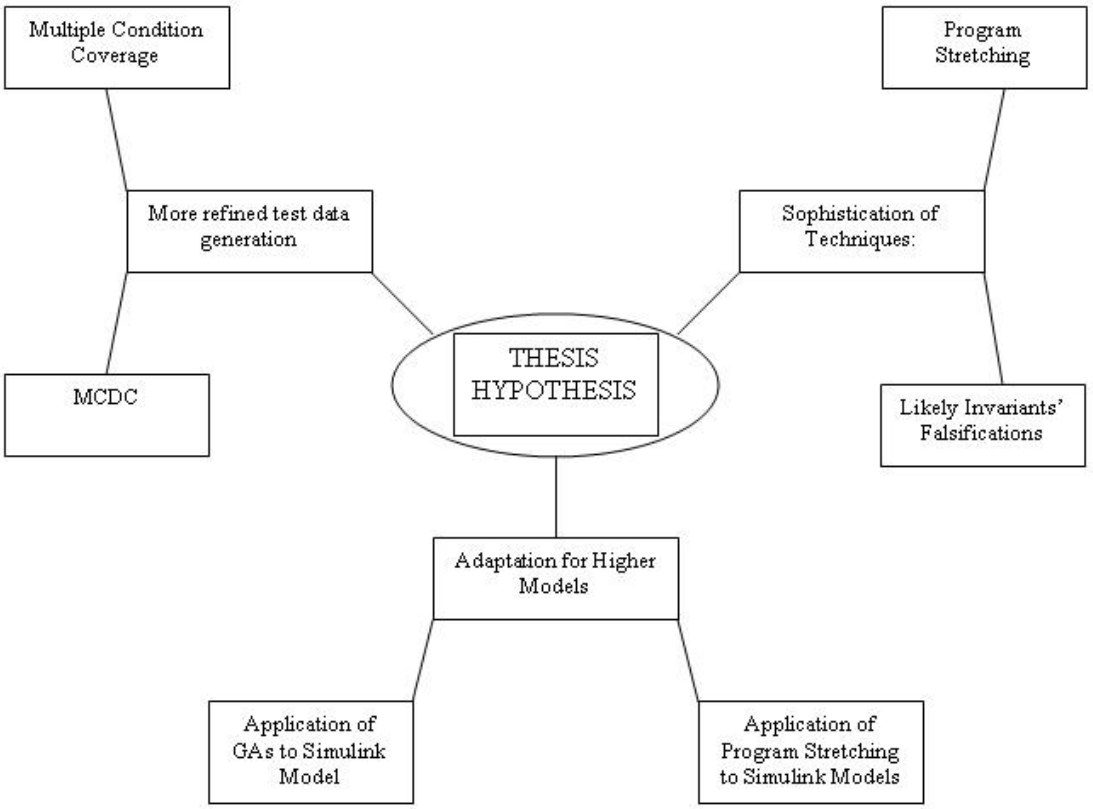


Figure 7.1: Thesis Hypothesis

### 7.1.1 Structural Testing: Searching for Fine Grained Test Data

In this work we proposed an instrumentation strategy coupled with the traditional search based approach to generate test data at a fine level of condition granularity. For the proof of concept multiple condition and modified condition decision coverage criteria were selected and simulated annealing was adapted as an optimisation technique. The proposed technique was evaluated with programs from the current software testing literature. Experiments were also carried out to optimise the parameters of simulated annealing.

Most of the time in search based test data generation the coverage criterion used is branch coverage. The work in this thesis is a demonstration of how search based techniques can be applied to generate test data for stronger criteria. We believe that this sophistication is important as many industrial standards such as those used in avionics mandate stronger criteria such as MC/DC [162] as a minimum for structural based testing.

We developed a framework to carry out the above mentioned work. However, it is not a completely automated solution. We concentrated on specific components. Thus, part of the framework i.e., selecting sub-paths to reach the decision under consideration is done manually. However, these processes can be automated as has already been demonstrated by many existing works [176, 186, 118].

### 7.1.2 Application of Genetic Algorithms to Simulink Models

In this work we provided sophistication by using genetic algorithms to generate test data for MATLAB Simulink models, extending the existing work by Zhan and Clark [196].

There is much application of search based techniques at code level. However, there is very little work at architectural level. We ought to be thinking of applying different search techniques at this level as well. We took a step forward for Simulink models and did comparative studies for two of the most commonly used optimisation techniques. We performed experiments with a set of models taken from existing research literature. In general GAs perform better, however, we are

not claiming that it is the best technique for Simulink models. We believe that to substantiate this claim further, more work is required and further techniques and their variants need to be assessed.

### 7.1.3 Program Stretching

In this work we proposed 'program stretching', a novel technique which targets test data generation for 'hard' branches. To evaluate the technique, we conducted case studies at the code as well as architectural level. We received mixed results. At the code level, we experimented with three programs with varying degrees of structural complexity. In the most complex program, we obtained an improvement in the average number of evaluations as well as in the success rate. At the architecture level, we applied it to three Simulink models. Two variants of the approach were evaluated. The first variant did not perform well. However, by slightly improving the technique, significant improvement in success rate occurred. But in case of Simulink models the technique was very expensive in terms of number executions required to find the test data.

Program stretching is in its initial stages. Limited case studies were conducted for the proof of the concept. We believe much further research is required to establish this technique. Still there are issues which need to be resolve such as the 'cooling schedule' for the auxiliary variables, identification of 'hard' branches and the branches to be transformed and cost evaluation as the technique can be very expensive. Further real world application is needed.

The general idea of 'program stretching' is innovative and the idea would appear to generalise. Thus, investigation of 'program stretching' seems an obvious way forward. We know of no similar work in the literature.

### 7.1.4 Strengthening Inferred Specifications using Search Based Testing

In this work, we proposed a framework to refine automatically generated likely invariants. For a proof of concept, we inferred likely specifications for five bench-

mark programs with different configurations, using the widely known Daikon invariant detector [60].

By its very nature Daikon infers assertions that are consistent with witnessed traces but which are not necessarily invariants. The targetting of such putative invariants by search based techniques demonstrates that search based testing approaches (ours, and by implication, those of others too) can contribute to a major problem in software engineering. Daikon is a powerful tool. Our work shows how search based approaches can make it even better.

Our work also draws attention to possible synergies between proof and testing. Many advanced techniques require formal specifications of behaviour. Daikon has already established the ability of test executions to contribute to the creation of useful specification fragments. Our invariant falsification work augments the Daikon tool to increase the quality of inferred predicates, thus facilitating possible subsequent formal verification. Furthermore, a more general use of our falsification approach could save time for those seeking formal proof of correctness. If a program is indeed flawed, failing to find a proof of correctness is inevitable and a very considerable effort. Formal proofs could be attempted only after significant attempts have been made to find counter-examples using search based approaches (probably in conjunction with other methods as appropriate). This is a potentially significant area for deployment of search based test data generation techniques.

## 7.2 Future Work

Some possible future work directions.

### 7.2.1 Extension to the Proposed Framework

The framework for fine grained test data generation was developed to provide a proof of concept. It can be extended to make it a completely automated testing tool. We can incorporate different techniques that have been proposed to overcome different limitations of search based testing. This will provide a platform to

better assess the true efficacy of search based test data generation techniques.

### 7.2.2 Application of Search Based Techniques to Simulink Models

Application of GAs can be extended to the state based and mutation testing of MATLAB Simulink models (Both issues were addressed with SA in the work of Zhan and Clark [197, 198]. The framework can also be extended to generate test data for more complex models by incorporating techniques for Stateflow analysis. Other search based techniques such as EDAs can be implemented for comparative studies. We also aim to investigate other models such as UML dynamic diagrams (e.g., sequence and collaboration diagrams). We are interested in whether test data can be generated using search based techniques for coverage criteria as proposed by Rountev et al. [161].

### 7.2.3 Program Stretching

The work can be also extended to other areas as given below:

**Real World Examples** The technique can be evaluated with more real world programs such as those used in automobile and avionics industry.

**Search Based Algorithm** Only simulated annealing was used to assess the technique. The effectiveness of other techniques such as GAs still needs to be investigated.

**Invariant Falsification** Suppose we have a proposed invariant *inv* for some identified point in the program. If we insert a program statement "if(!inv);" at that point we can view the falsification of the invariant as a branch reachability problem. As a consequence we can easily apply the program stretching technique.

**Exception Generation** Consider assignment statements of the form "var=expr;". When expr is evaluated the result must be a value inside the type bounds, if an exception is not to be raised. In program reasoning it is common to refer

to such constraints as healthiness pre-conditions. We can insert a statement of the form "if(!healthiness); and proceed much as before.

**Higher Level Models** In this work we attempted application of program stretching to the 'path coverage' or more specifically to the 'combination' coverage [196] of Simulink models. Zhan and Clark [197] also proposed techniques for a more practical branch coverage criterion. We could extend the approach to such criteria. We also believe that our stretching technique may find application to other higher level models such as statecharts. Similarly, automated SBTDG from specification might be facilitated.

### 7.2.4 Likely Invariants' Falsification

We could also seek complete automation of the process of invariant falsification. This includes automatic generation of the *invariants' class*. One way this can be achieved is by extending the 'Annotate tool' in Daikon tool set. Further work can also be carried out to solve the 'automatic' oracle problem, especially for legacy software or software for which specifications do not exist. In fact, search based test data generation techniques can be used to generate the initial test suite, including boundary test values. This may give us a more refined set of inferred invariants, which can be subjected to further refinement using the technique described in Chapter 6. Furthermore, we aim to carry out a statistical comparison of existing techniques with our proposed technique for deriving a refined set of likely inferred invariants. Techniques will be compared for efficiency in term of computation cost and derivation of correct sets of likely invariants.

## 7.3 Conclusion

The work in thesis has shown how current approaches to search based test data generation can be refined and extended. The fine-grained criterion satisfaction work generally seems a natural extension of the wealth of code-oriented search based test data generation work in the research literature. The comparison of optimisation techniques in the context of Simulink models makes a modest contribution

to an area where little research has been performed. Program Stretching seems a highly innovative idea with wide potential applicability. Finally, the application of search based test data generation techniques to invariant falsification shows how these techniques can complement the best available techniques in a very important domain.

Appendix A

# Triangle Program with Instrumented Version

```
1  public class Triangle {
2    public void triangleType(int x, int y, int z){
3      if (x<=0||y<=0||z<=0)
4      {
5        System.out.println("Invalid inputs: negative or zero side");
6      }
7      else if (x>y+z||y>x+z||z>x+y)
8      {
9        System.out.println("Invalid inputs: sum of two side is less
             than the third side");
10     }
11        else if( x == y && y == z) {
12           // triangle is equilateral
13           System.out.print( "an equilateral " );
14         } else if( x == y || y == z || x == z ) {
15           // triangle is isoceles
16           System.out.print( "an isoceles " );
17         } else {
18           // triangle is scalene
19           System.out.print( "a scalene " );
20         }
21   }
22 }
```

## A.1 Instrumented Version of Triangle Program

```
1  public class Triangle implements Serializable {
2    public Data data = new Data(4, 2, 1);
3
4    public void triangleType(int x, int y, int z)  {
5      if (data.complex(0, data.basic(x, "<=", 0, 0, 0, 0) | data.basic(
           y, "<=", 0, 0, 1, 0) | data.basic(z, "<=", 0, 0, 2, 0))) {
6        System.out.println("Invalid inputs: negative or zero side");
7      }else
8        if (data.complex(1, data.basic(x, ">", y + z, 1, 0, 0) | data.
             basic(y, ">", x + z, 1, 1, 0) | data.basic(z, ">", x + y, 1,
              2, 0))) {
9          System.out.println("Invalid inputs: sum of two side is less
               than the third side");
10       }else
11         if (data.complex(2, data.basic(x, "==", y, 2, 0, 0) & data.
               basic(y, "==", z, 2, 0, 1))) {
12           System.out.print("an equilateral ");
13         }else
14           if (data.complex(3, data.basic(x, "==", y, 3, 0, 0) | data.
                 basic(y, "==", z, 3, 1, 0) | data.basic(x, "==", z, 3,
                 2, 0))) {
15             System.out.print("an isoceles ");
16           }else {
17             System.out.print("a scalene ");
18           }
19
20   }
21 }
```

Appendix B

# Programs used in this thesis work

### B.0.1 CalDate

```
1  public class CalDate {
2    public double toJulian ( int day , int month , int year ) {
3      int JGREG = 15+31*(10+12*1582);
4      double HALFSECOND = 0.5 ;
5      int julianYear = year ;
6      if ( year <0){ julianYear = julianYear+1;
7       }
8      int julianMonth = month ;
9      if (month>2){
10     julianMonth = julianMonth+1;
11     } else {
12     julianYear = julianYear-1;
13
14     julianMonth = julianMonth+13;
15    }
16     double t = Math.floor(365.25* julianYear ) ;
17     double s = Math.floor (30.6001* julianMonth ) ;
18     double julian = t+s+day+1720995.0;
19     int temp = day+31*(month+12*year ) ;
20     if ( temp>=JGREG){
21     // change over to Gregorian cal endar
```

```
22      int ja = ( int ) ( 0.01* julianYear ) ;
23      julian = julian+2-ja +(0.25* ja ) ;
24      }
25     return Math.floor(julian ) ;
26     }
27 }
```

### B.0.2 Quadratic

```
1  public class Quadratic {
2    public void quad(int a, int b, int c){
3
4      if (a==0){
5        System.out.println("Not a quadratic equation");
6
7      }
8      else if((b*b-4*a*c)>0){
9          System.out.println("Roots are real and unequal");
10       }
11       else if ((b*b-4*a*c)==0){
12         System.out.println("Roots are real and equal");
13       }
14       else {
15         System.out.println("Roots are complex");
16       }
17     }
18 }
```

### B.0.3 Expint

```
1  public class Expint {
2    public static final int MAXIT = 100;
3    public static final double EULER = 0.5772156649;
4    public static final double FPMIN = Math.exp(-30);
5    public static final double EPS = Math.exp(-7);
6
7    public double expint(int n, double x){
```

```
8       int i, ii, nm1;
9       double a,b,c,d,del,fact,h,psi,ans = 0;
10      nm1=n-1;
11      if(n<0||x<0.0 || (x==0.0)&&(n==0||n==1)){
12        System.out.println("Bad arguments in expint");
13      }
14      else
15      {
16        if(n==0)
17          ans=Math.exp(-x)/x;
18        else {
19          if (x==0.0)
20            ans=1.0/nm1;
21          else
22            if(x>1.0){
23              b=x+n;
24              c=1.0/FPMIN;
25              d=1.0/b;
26              h=d;
27              for (i=1;i<=MAXIT;i++){
28                a=-i*(nm1+i);
29                b+=2.0;
30                d=1.0/(a*d+b);
31                c=b+a/c;
32                del=c*d;
33                h*=del;
34                if (Math.abs(del-1.0)<EPS){
35                  ans=h*Math.exp(-x);
36                  return ans;
37                }
38
39              }
40              System.out.println("continued function failed in expint")
                    ;
41            }else {
42              if(nm1!=0){
43                ans=1/nm1;
44              }
```

```
45              else{
46                ans=-Math.log(x)-EULER;
47              }
48
49              fact=1.0;
50              for (i=1;i<MAXIT;i++){
51                fact*=-x/i;
52                if (i!=nm1)
53                  del=-fact/(i-nm1);
54                else{
55                  psi=-EULER;
56                  for (ii=1;ii<=nm1;ii++)
57                    psi+=1.0/ii;
58                  del=fact*(-Math.log(x)+psi);
59                }
60                ans+=del;
61                if(Math.abs(del)<Math.abs(ans)*EPS)
62                  return ans;
63              }
64
65            }
66        }
67
68      }
69      return ans;
70    }
71  }
```

### B.0.4 Complex or Program3 for program stretching, code based

```
1  public class Complex{
2    public void complexCheck(int a, int b, int c,int d, int e)  {
3      if ((a<10) && (b>1000)&&(c>999  )) {
4        System.out.println("complex8 executed");
5      }else if ((a>10) && (b>1000)&& (c>999)&& (d>999&& d<1001)) {
6        System.out.println("complex8 else if executed");
7        if (c+b<2003&& a+b<1400&&a>390) {
```

```
8           System.out.println("the inner if executed");
9           if (d+e>2001 && d+e<2003)
10          {
11            System.out.println("Target reached");
12          }
13        }
14      }else {
15        System.out.println("the false branch taken");
16      }
17    }
18  }
```

```
1  public class Middle implements Serializable {
2    public int findMiddle(int a, int b, int c){
3
4
5      if((a<b && b<c)|| (c<b && b<a)){
6        System.out.println("middle value is b="+b);
7        return b;
8      }
9      else  if((a<c && c<b)||(b<c && c<a )){
10       //System.out.println("middle value is c="+c);
11       return c;
12     }
13     else if((b < a && a <c) || (c<a && a<b) ){
14       //System.out.println("middle value is a="+a);
15       return a;
16     }
17     else if (b==c){
18       //System.out.println(" b is equal to c,  middle value is"+a);
19       return a;
20     }
21     else if (b==a){
22       //System.out.println(" b is equal to a,  middle value is"+c);
23       return c;
24
25     }
26     else {
```

```
27        //System.out.println(" a is equal to c" );
28        return b;
29      }
30    }
31 }
```

```
1  public class WrapRoundCounter implements Serializable{
2
3    public int wrap_inc(int n){
4      if (n>10)
5      {
6        n=0;
7        return n;
8
9      }
10     else
11     {
12       n=n+1;
13       return n;
14     }
15   }
16 }
```

```
1  public class BubbleSort {
2    public void sort( int array[] )
3    {
4      // loop to control number of passes
5      for ( int pass = 1; pass < array.length; pass++ ) {
6
7        // loop to control number of comparisons
8        for ( int element = 0;
9        element < array.length - 1;
10       element++ ) {
11
12         // compare side-by-side elements and swap them if
13         // first element is greater than second element
14         if ( array[ element ] > array[ element + 1 ] )
15           swap( array, element, element + 1 );
```

```
16
17        } // end loop to control comparisons
18
19      } // end loop to control passes
20
21    } // end method bubbleSort
22
23    // swap two elements of an array
24    public void swap( int array3[], int first, int second )
25    {
26      int hold;  // temporary holding area for swap
27
28      hold = array3[ first ];
29      array3[ first ] = array3[ second ];
30      array3[ second ] = hold;
31    }
32
33 } // end class BubbleSort
```

### B.0.5 Program 1

```
1  public class Program1  {
2
3
4    public void opCheck(int j, int k,
5                        int l, int m){
6
7       if ((j>10) && (j< 15)){
8
9         System.out.println("j>10 && j<15: executed");
10
11        if (k== j) {
12
13          System.out.println("k=j:executed");
14
15          if ((l!=k) && (m>=j) &&(m==k)) {
16
```

```
17                System.out.println("Target Reached"); .....(1)
18               }
19            }
20          }
21      }
22  }
```

**Transformed Program 1**

```
1  public class Program1Var {
2
3     public void opCheck(int x, int y,
4                         int z,int m,
5                         int var1,int var2,
6                         int var3,int var4){
7
8       if (x+var1>10 && x-var2<15){
9
10        System.out.println("x>10 && x<15: executed");
11
12          if (y+var3>=x&&x+var3>=y){
13
14             System.out.println("y==x:executed");
15
16             if (z !=y && m>=x && m==y){
17
18               System.out.println("Target reached");
19             }
20          }
21       }
22   }
23  }
```

### B.0.6 Program 2

```
1  public class Program2 {
2
```

```
3     public void complexCheck(int x, int y, int z) {
4        if ((x<10) &&(y>1000) && (z>999)) {
5
6           System.out.println("Branch 1 executed");
7        }
8    else  if ((x > 10) &&(y > 1000) && (z> 999)){
9
10          System.out.println("Branch 1 else-if executed");
11
12          if (((z + y) < 2003) && (x + y< 1400) && (x > 390)) {
13             System.out.println
14            ("Target reached");.....(2)
15
16          }
17       }
18   else {
19
20       System.out.println("the false branch taken");
21       }
22    }
23 }
```

**Transformed Program 2**

```
1  public class Program2Var {
2
3     public void complexCheck(int x, int y,
4                              int z, int var1,
5                              int var2,int var3){
6
7        if ((x<10) &&(y>1000)
8                               && (z>999)){
9
10          System.out.println("Branch-1 executed");
11
12       }
13
```

```
14        else if ((x+var1 > 10) &&(y+var2 > 1000)
15                                              &&(z+var3> 999)){
16
17             System.out.println("Branch-1 else-if executed");
18
19             if (((z + y) < 2003) && (x + y< 1400)
20                                          && (x > 390)){
21
22                System.out.println
23               ("Target reached");
24             }
25          }
26        else  {
27
28           System.out.println("False branch taken");
29        }
30    }
31 }
```

### B.0.7 Program 3

```
1  public class Program3 {
2
3    public void complexCheck(int a, int b,
4                             int c, int d, int e)  {
5
6        if ((a<10) && (b>1000)&&(c>999  )) {
7
8          System.out.println("Branch-1 executed");
9
10       }
11       else if ((a>10) && (b>1000)&& (c>999)
12                                    && (d>999&& d<1001)) {
13
14         System.out.println("Branch-1 else if executed");
15
16         if (c+b<2003&& a+b<1400&&a>390) {
```

```
17
18              System.out.println("Inner if executed");
19
20              if (d+e>2000 && d+e<2003){
21
22                 System.out.println
23               ("Target reached");.....(3)
24             }
25          }
26      }
27         else {
28
29           System.out.println("False branch taken");
30         }
31    }
32 }
```

**Transformed Program 3**

```
1  public class Program3Var {
2
3     public void complexCheck(int a, int b,
4                         int c,int d,
5                         int e, int var1,
6                         int var2,int var3,
7                         int var4,int var5,
8                  int var6)  {
9
10       if ((a<10) &&(b> 1000) &&(c> 999)) {
11
12          System.out.println("Branch-1 executed");
13
14       }
15       else if (((a+var1> 10) &&
16                         (b+var2 >1000)) &&
17                          c+var3> 999)&&
18                         (d+var4 >999)&&
```

```
19                          (d<1001)) {
20
21            System.out.println("Branch-1 else if executed");
22
23            if ((c + b<2003+var5) &&
24             (a + b<1400)&&(a+var6>390)){
25
26               System.out.println("Inner if executed");
27
28               if ((d + e> 2001) && (d + e<2003)) {
29
30                  System.out.println("Target reached");
31               }
32            }
33         }
34         else {
35            System.out.println("False branch taken");
36         }
37  }
38 }
```

## B.1 Models for Test data generation using GAs for Simulink Models

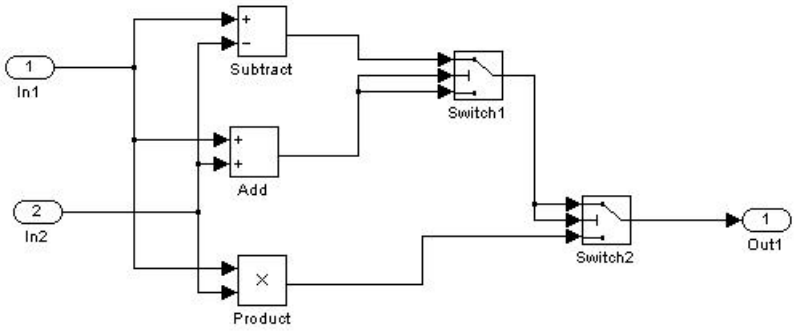


Figure B.1: SmplSW

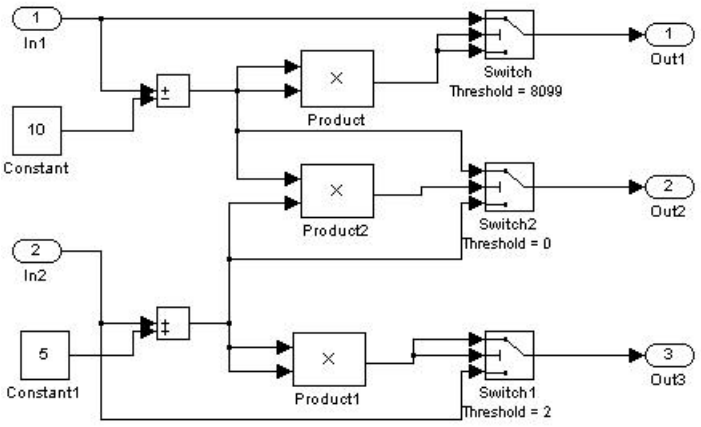


Figure B.2: Quadratic v1

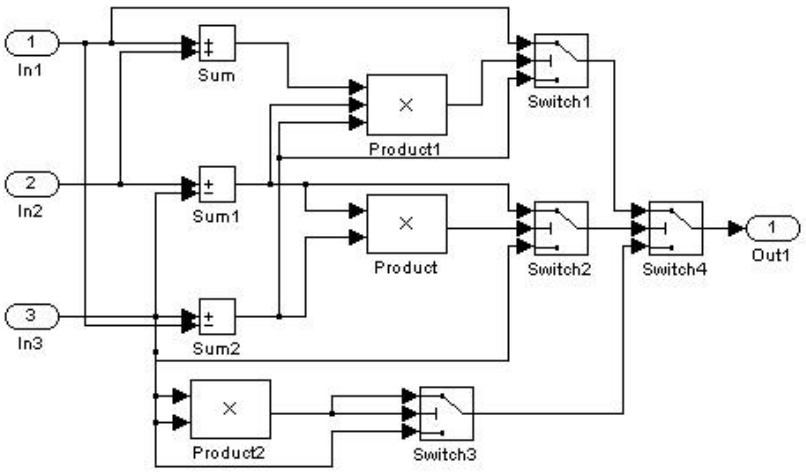


Figure B.3: RandMdl

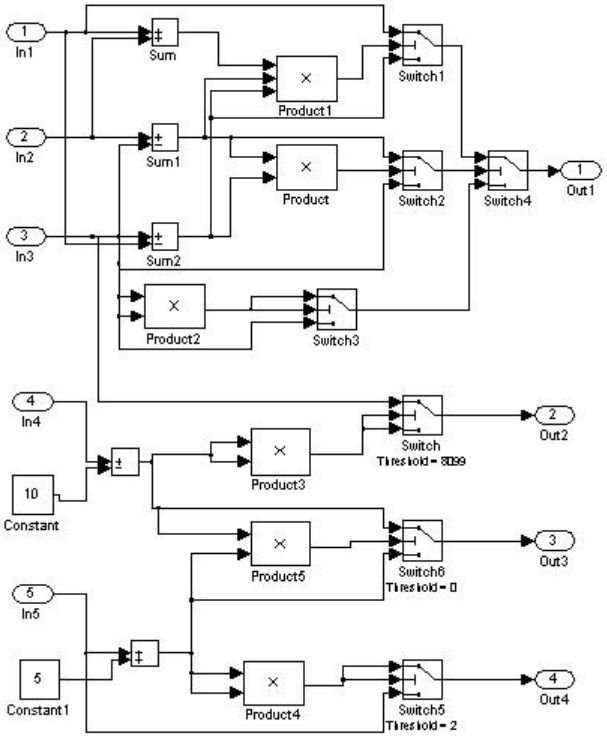


Figure B.4: Combine

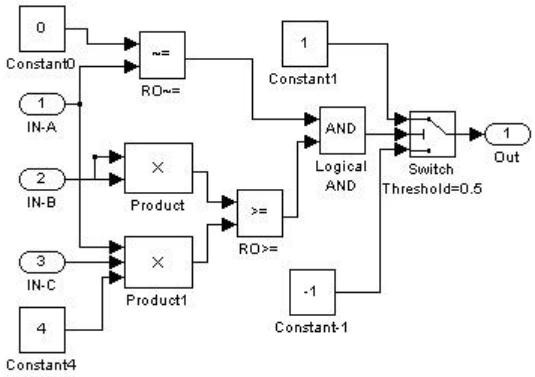


Figure B.5: Quadratic v2

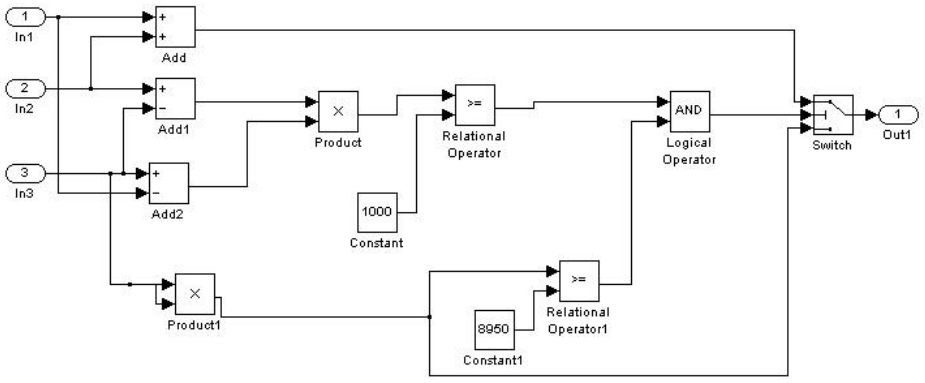


Figure B.6: Tiny

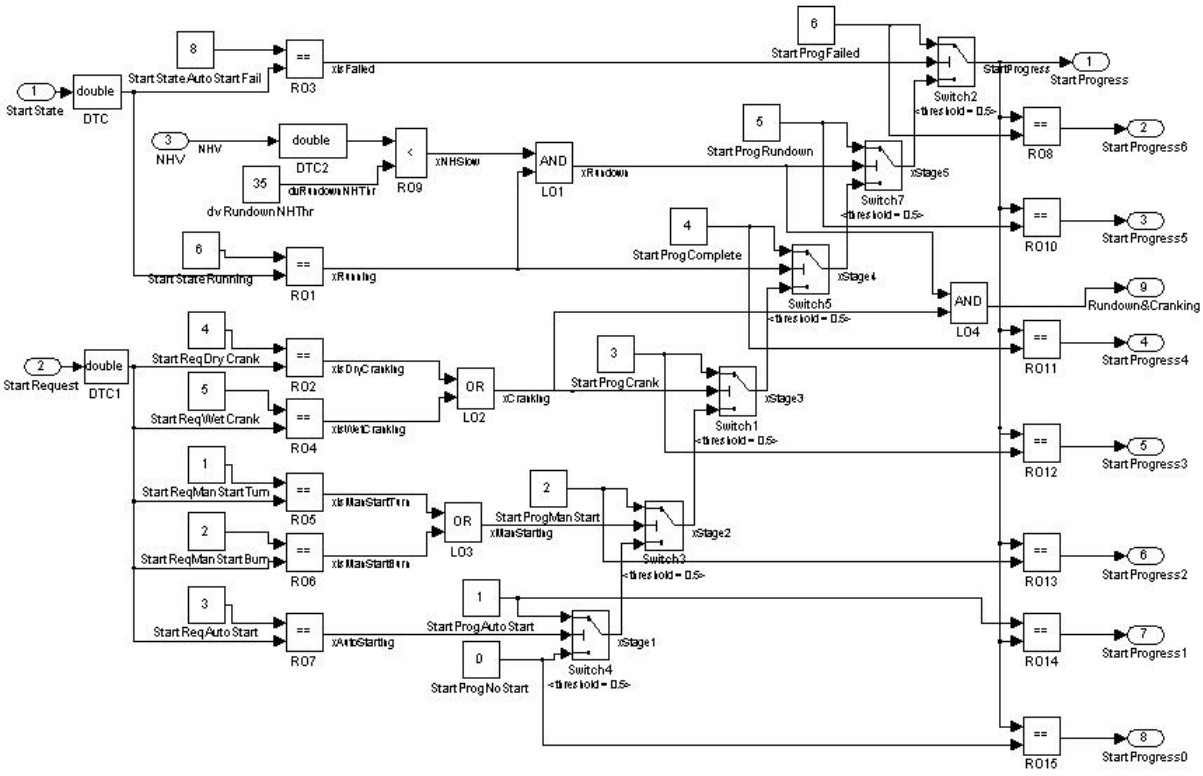


Figure B.7: Calc-Start-Progress

# Bibliography


[1] ANTLR parser generator, http://www.antlr.org.

[2] Genetic Algorithm and Direct Search Toolbox, The MathWorks Inc. http://www.mathworks.com/products/gads/.

[3] JTest, ParaSoft Corportation, http://www.parasoft.com.

[4] MATLAB Real-Time Workshop 7.4. online (http://www.mathworks.co.uk/products/rtw/ ).

[5] Object Management Group. http://www.omg.org/.

[6] Real-time workshop embedded coder 5.4. online (http://www.mathworks.com/products/rtwembedded/).

[7] UML Testing Profile version 1.0. www.omg.org, July 2005.

[8] Daikon invariant detector user's manual. (online) http://groups.csail.mit.edu/pag/daikon/, May 2007. Daikon version 4.2.16.

[9] Meta Object Facility (MOF) 2.0 Query/View/Transformation Specifications. http://www.omg.org/spec/QVT/1.0/PDF, April 2008.

[10] A. Abdurazik, P. Ammann, Wei Ding, and J. Offutt. Evaluation of three specification-based testing criteria. In *proceedings of the 6th IEEE International Conference on Complex Computer Systems (ICECCS '00)*, pages 179–187, Tokyo, Japan, September 2000. IEEE Computer Society.

[11] A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *proceedings of The Third International Conference on the Unified Modeling Language (UML '00)*, pages 383–395, York, UK, October 2000.

[12] A.Gargantini and E.Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science, JUCS*, 10(8), 2001.

[13] A.J.Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *proceedings of fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999.

[14] E. Alba and J.F. Chicano. Software testing with evolutionary strategies. In *The 2nd Workshop on Rapid Integration of Software Engineering Techniques (RISE05)*, volume 169, pages 50–65., Heraklion, Greece, September 2005.

[15] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[16] Scott W Amber. UML 2 Class Diagrams. online (http://www.agilemodeling.com/artifacts/classDiagram.htm).

[17] Paul Ammann and Jeff Offutt. Using Formal Methods To Derive Test Frames In Category-Partition Testing. In *proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS 94)*, pages 69–80, Gaithersburg, MD, USA, 27 Jun-1 Jul 1994.

[18] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing Verification and Reliability*, 13(2):95–127, 2003.

[19] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th interna-*

*tional conference on Software engineering*, pages 402–411, New York, NY, USA, 2005. ACM.

[20] Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. MC/DC automatic test input data generation. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation*, pages 1657–1664, New York, NY, USA, 2009. ACM.

[21] Andre Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In *proceedings of Genetic and Evolutionary Computation Conference (GECCO)*, July 2003.

[22] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[23] Walid Ben-Ameur. Computing the initial temperature of simulated annealing. *Comput. Optim. Appl.*, 29(3):369–385, 2004.

[24] Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison Wesley, 2000.

[25] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Software Testing, Verification and Reliability*, 11(1):39–54, 2001.

[26] L. Bottaci. Predicate expression cost functions to guide evolutionary search for test data. In *proceeding of the Genetic and Evolutionary Computation Conference (GECCO03)*, pages 2455–2464, 2003.

[27] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[28] Lionel C. Briand and Yvan Labiche. A UML-Based Approach to System Testing. In *proceedings of the 4th International Conference on The Unified*

*Modeling Language, Modeling Languages, Concepts, and Tools*, pages 194–208, London, UK, 2001. Springer-Verlag.

[29] Adriana Carniello, Mario Jino, and Marcos L.Chaim. Structural testing with use cases. In *Workshop on Requirements Engineering*, Tandil, Argentina, December 2004.

[30] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. In Jonathan P. J. Bowen, editor, *Z User Workshop*, pages 51–68, Cambridge, UK, 1994. Springer-Verlag.

[31] Steve Cayzer. Artifical immune systems. (online) http://www.hpl.hp.com/personal/Steve_Cayzer/downloads/050210Bristol_draft7a.pdf, 2005.

[32] John Joseph Chilensky and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9 (5):193–200, 1994.

[33] B.J. Choi, R.A. DeMillo, E.W. Krauser, R.J. Martin, A.P. Mathur, A.J. Offutt, H. Pan, and E.H. Spafford. The mothra tool set (software testing). In *System Sciences, 1989. Vol.II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 2, pages 275–284 vol.2, Jan 1989.

[34] Byoungju Choi and Aditya P. Mathur. High-performance mutation testing. *J. Syst. Softw.*, 20(2):135–152, 1993.

[35] T.S. Chow. Testing Software Design Modelled by Finite State Machine. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[36] John A. Clark, Jose Javier Dolado, Mark Harman, Bryan Jones, Mary Lumkin, Brian Mitchell, S. Mancoridis, Kearton Rees, and Marc Roper. Reformulating software engineering as a search problem. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[37] L. A. Clarke, A. Podgurski, and S. J Richardson, D. J.and Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transaction on Software Engineering*, 15(11):1318–1332, November 1989.

[38] Fred Clover and Gary A Kochenberger, editors. *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*. Springer, 2003.

[39] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *proceedings of the 5th ACM Symposium on Principles of Programming Languages POPL'78*, Tuscan, Arizona United States, January, 1978.

[40] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for java. *Software Practice and Experience*, 34(11):1025–1050, Sept. 2004.

[41] Elspeth Cusack and Clazien D. Wezeman. Deriving Tests for Objects Specified in Z. In *proceedings of the Z User Workshop*, pages 180–195, London, UK, 1992. Springer-Verlag.

[42] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *proceedings of the 2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, CA, USA., October 2003.

[43] Eugenia Daz, Javier Tuya, Raquel Blanco, and Jos Javier Dolado. A tabu search algorithm for structural software testing. *Computers & Operations Research*, 35(10):3052–3072, 2008.

[44] Leandro Nunes de Castro and Jonathan Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002.

[45] D. de Werra and A. Hertz. Tabu search techniques. *OR Spectrum*, 11(3):131–141, 1989.

[46] W. H. Deason, D. B. Brown, K. H. Chang, and J. H. Cross II. A rule-based software test data generator. *IEEE Trans. on Knowl. and Data Eng.*, 3(1):108–117, 1991.

[47] Harvey M. Deitel and Paul J. Deitel. *Java How to Program.* Prentice Hall, December 16, 2002.

[48] E. Delamaro, Jose C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on software Engineering*, 27(3):228–247, 2001.

[49] R.A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[50] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducass. Proving or disproving likely invariants with constraint reasoning. In *proceedings of the 15th Workshop on Logic-based Methods in Programming Environments (WLPE'05)*, Sitges Barcelona, Spain, October 5 2005.

[51] David L Detlefs, K. Rustan M Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159 SRC-RR-159, Compaq Systems Research Center, December 1998.

[52] Eugenia Diaz, Javier Tuya, and Raquel Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 310 – 313, Oct. 2003.

[53] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[54] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization. *Artif. Life*, 5(2):137–172, 1999.

[55] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart. Test Data Generation from UML State Machine Diagrams using GAs. In *proc. International Conference on Software Engineering Advances ICSEA 2007*, pages 47–47, 2007.

[56] J. W. Duran and S. C. Ntafos. Evaluation of random testing. *IEEE Transactions on Software Engineering. Vol. SE-10, no. 4, pp. 438-444. 1984*, 10(4):438–444, 1984.

[57] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering.* Pearson Education Limited, London, 2003.

[58] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, August 2000.

[59] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, New York, NY, USA, 1999. ACM.

[60] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthews S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45, 2007.

[61] Mark Fawster and Dorothy Graham. *Software Test Automation.* Addison-Wesley Professional, 1999.

[62] Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[63] P. G. Frankl and J. E Weyuker. An applicable family of data flow testing criteria. *IEEE Transaction of Software Engineering*, 14:14831498, 1988.

[64] M. R. Girgis. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithms. *Journal of Universal Computer Science*, 11(6):898–915, 2005.

[65] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549, 1986.

[66] Fred Glover and Manuel Laguna. Tabu search. web page.

[67] Fred Glover, Eric Taillard, and Dominique de Werra. A user's guide to tabu search. *Ann. Oper. Res.*, 41(1-4):3–28, 1993.

[68] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. pages 69–93, 1991.

[69] Hans-Gerhard Gross. Testing and UML, A Perfect Fit. Technical Report 110.03/E, Version 1.0, IESE, Oct 2003.

[70] Neelam Gupta. Generating test data for dynamically discovering likely program invariants. In *proceedings of Workshop on Dynamic Analysis (WODA 2003)*, pages 21–24, Portland, Oregon, May 9, 2003.

[71] Neelam Gupta and Rajiv Gupta. *Data Flow Testing in The Compiler Design Handbook, Optimization and Machine Generation Code.* CRC Press, September 2002.

[72] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8-10, 2003.

[73] Agrawal H, Richard A. Demillo, Bob Hathaway, William Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, Aditya P. Mathur, and Eugene Spafford.

Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University., 1989.

[74] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[75] M. Harder, J. Mellen, and Michael D Ernst. Improving test suites via operational abstraction. In *proceedings of the 25th International Conference on Software Engineering*, pages 60–71, 2003.

[76] M. Harman, L. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO 2002).*, 2002.

[77] Mark Harman. The current state and future of Search Based Software Engineering. In Lionel Briand and Alexander Wolf, editors, *Future of Software Engineering 2007*, pages 342–357, Los Alamitos, California, USA, 2007.

[78] Mark Harman. Open problems in testability transformation. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 196–209. IEEE Computer Society, 2008.

[79] Mark Harman, Andr Baresel, David Binkley, Robert Hierons, Lin Hu, Bogdan Korel, Phil McMinn, and Marc Roper. *Formal Methods and Testing*, volume 4949 of *Lecture notes in Computer Science*, chapter Testability Transformation Program Transformation to Improve Testability, pages 320–344. Springer Berlin / Heidelberg, April 2008.

[80] Mark Harman, Lin Hu, Rob Hierons, Malcolm Munro, Xingyuan Zhang, Jose Javier Dolado, Mari Carmen Otero, and Joachim Wegener. A Post-Placement Side-Effect Removal Algorithm. In *proceedings of IEEE Interna-*

*tion conference on Software Maintenance (ICSM 2002)*, pages 2–11. IEEE Computer Society Press, 2002.

[81] Mark Harman, Lin Hu, Robert Mark Hierons, Joachim Wegener, Harmen Sthamer, Andr Baresel, and Marc Roper. Testability Transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[82] Mark Harman, Malcolm Munro, Lin Hu, and Xingyuan Zhang. Side-effect removal transformation. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*, pages 310–319. IEEE Computer Society, 2001.

[83] Jim Heumann. Generating Test Cases from Use cases, 2001. Rational Software.

[84] R. M. Hierons, M. Harman, and C. J. Fox. Branch-coverage testability transformation for unstructured programs. *Comput. J.*, 48(4):421–436, 2005.

[85] Robert M. Hierons. Testing from a Z specification. *Softw. Test., Verif. Reliab.*, 7(1):19–33, 1997.

[86] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[87] Seok Hong, Young Gon Kim, Sung Deok Cha, Doo Hwan Bae, and Hasan Ural. A test sequence selection method for statecharts. *Software Testing,Verification and Reliability*, 10(4):203–227, 2000.

[88] Hans-Martin Hörcher. Improving Software Tests Using Z Specifications. In *ZUM '95: Proceedings of the 9th International Conference of Z Usres on The Z Formal Specification Notation*, pages 152–166, London, UK, 1995. Springer-Verlag.

[89] WE Howden. Weak Mutation Testing and Completeness of Test Sets. *IEEE. Transactions on Software Engineering*, 8:371–379, 1982.

[90] IEEE. *Guide to software engineering body of knowledge*, pages 5–1. 2004.

[91] The MathWorks Inc. http://www.mathworks.com/.

[92] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[93] Abdul Salam Kalaji, Robert Mark Hierons, and Stephen Swift. Generating feasible transition paths for testing from an extended finite state machine (efsm). *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:230–239, 2009.

[94] AbdulSalam Kalaji, Robert Mark Hierons, and Stephen Swift. A Testability Transformation Approach for State-Based Programs. *Search Based Software Engineering, International Symposium on*, 0:85–88, 2009.

[95] Cem Kaner. Measurement of the extent of testing. In *Pacific Northwest Software Quality Conference*, Portland, OR, USA, 2000.

[96] Sunwoo Kim, John A. Clark, and John A. Mcdermid. The rigorous generation of java mutation operators using hazop. In *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, 1999.

[97] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[98] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[99] J. Knight and P. Ammann. An experimental evaluation of simple methods for seeding program errors. In *proceedings of the Eighth International Conference on Software Engineering*, pages 337–342, London, UK, August 1985.

[100] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16:8:870–879, 1990.

[101] E.W. Krauser, A.P. Mathur, and V.J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[102] Ivan Kurtev. State of the art of QVT: a model transformation language standard. *Lecture Notes in Computer Science*, 5088/2008:377–393, 2008.

[103] Kiran Lakhotia, Mark Harman, and Phil McMinn. Handling dynamic data structures in search based testing. In *proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1759–1766, Atlanta, USA, July 2008. ACM press.

[104] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated Boundary Testing from Z and B. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 21–40, London, UK, 2002. Springer-Verlag.

[105] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/java users manual. Technical report 2000-002, Compaq Systems Research Center, Palo Alto, California, October 12 2000.

[106] Huaizhong Li and Chiou Peng Lam. Software test data generation using ant colony optimization. In *proceedings of the International Conference on Computational Intelligence*, pages 1–4, 2004.

[107] Konstantinos Liaskos and Marc Roper. Automatic test-data generation: An immunological approach. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 77–81, Washington, DC, USA, 2007. IEEE Computer Society.

[108] Sean Luke. *Essentials of Metaheuristics*. 2009. available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[109] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An Automated Class Mutation System. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[110] Vittorio Maniezzo, Luca Maria Gambardella, and Fabio De Luigi. Ant colony optimization. In *Optimization Techniques in Engineering. Springer-Verlag*, pages 101–117. Addison-Wesley, 2004.

[111] Nashat Mansour and Miran Salame. Date generation for path testing. *sotware Quality Journal*, 12:121–136, 2004.

[112] Brian Marick. *The Craft of Software Testing.* Prentice Hall, 1995.

[113] Aditya P. Mathur. *Foundation of Software Testing draft V2.11.* 2006.

[114] A.P. Mathur and E.W. Krauser. Mutant Unification for Improved Vectorization. Technical Report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN, April 1988.

[115] Peter S May. *Test Data Generation: Two Evolutionary Approaches to Mutation Testing.* PhD thesis, Kent University, UK, May 2007.

[116] McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[117] P. Mcminn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[118] Phil McMinn. IGUANA: Input generation using automated novel algorithms. A plug and play research tool. Technical report, Department of Computer Science, University of Sheffield, 2007.

[119] Phil McMinn, David Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18(3):1–27, 2009.

[120] Phil McMinn and Mike Holcombe. The State Problem for Evolutionary Testing. In *proceedings of the 2003 Conference on Genetic and Evolutionary Computation (GECCO '03)*, volume 2724, pages 2488–2498. Springer, 2003.

[121] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020, New York, NY, USA, 2005. ACM.

[122] Jacqueline A. McQuillan and James F. Power. A survey of UML-Based Coverage Criteria for Software Testing. Technical Report NUIM-CS-TR-2005-08, National University of Ireland, Maynooth, Co. Kildare, 2005.

[123] Thomas McTavish and Diego Restrepo. *Applications of Computational Intelligence in Biology*, chapter Evolving Solutions: The Genetic Algorithm and Evolution Strategies for Finding Optimal Parameters, pages 55–78. Studies in Computational Intelligence. Springer Berlin / Heidelberg, 2008.

[124] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[125] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chem. Phys*, 21:1087–1091, 1953.

[126] A. Metzger. *Model-driven Software Development*, chapter A systematic look at model transformations, pages 19–33. Springer, 2005.

[127] Christoph Michael and Gary McGraw. Automated software test data generation for complex programs. In *proceedings of the 13th IEEE international conference on Automated software engineering (ASE '98)*, pages 136–146. IEEE, 1998.

[128] Christoph C. Michael, Gary McGraw, Michael Schatz, and C. C. Walton. Genetic Algorithms for Dynamic Test Data Generation. In *Automated Software Engineering*, pages 307–308, 1997.

[129] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springe, 1996.

[130] W Miller and D.L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, Sept. 1976.

[131] H.D. Mills. On the statistical validation of computer programs. Technical Report FSC 72-6015, IBM, 1972.

[132] John Mitchell, Terence Parr, John Lilley, Scott Stanchfield, Markus Mohnen, Peter Williams, Allan Jacobs, Steve Messick, and John Pybus. Java parser and tree parser (grammar version 1.22 -april 14, 2004). http://www.antlr2.org/grammar/java, 2004.

[133] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[134] Ivan Moore. Jester. online: http://jester.sourceforge.net/.

[135] Tafline Murnane and Karl Reed. On the effectiveness of mutation analysis as a black box testing technique. In *proceedings of the 13th Australian Conference on Software Engineering, AWSEC01*, 2001.

[136] Glenford Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[137] F. Nagoya, S. Liu, and Y. Chen. Design of a Tool for Specification-Based Program Review. In *proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 10–11. IEEE Computer Society, 2005.

[138] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22-24, 2002.

[139] The Institute of Electrical and Electronics Engineers Inc. Standard glossary of software engineering terminology (ansi), 1991.

[140] A. J. Offut. A practical system for mutation testing: Help for the common programmer. In *proceedings of Twelfth International Conference on Testing Computer Software*, 1995.

[141] A.J. Offut, R. Pargas, S.V. Fichther, and P. Khambekar. Mutation testing of software using a mimd computer. In *proceedings of International Conference on Parallel Processing*, pages 257–266, Chicago, USA, August 1992.

[142] A. J. Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165192, September 1997.

[143] A. J. Offutt, G. Rothermel, and C. Zapf. An experimenetal evaluation of selective mutation. In *proceedings of the Fifteenth International Conference on Software Engineering*, Baltimore, MD, May 1993.

[144] A.J. Offutt. Generating Test Data From Requirements/Specifications. Technical Report ISSE-TR-98-01, George Mason University, March 1998.

[145] A.J. Offutt and Roland Untch. 2000: Uniting the Orthogonal. In *proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[146] J. Offutt and A. Abdurazik. Generating tests from UML specifications. *Lecture notes in computer science*, pages 416–429, 1999.

[147] T. Okabe, Y. Jin, and B. Sendhoff. Theoretical comparisons of search dynamics of genetic algorithms and evolution strategies. In *Proc. IEEE Congress on Evolutionary Computation*, volume 1, pages 382–389 Vol.1, 2005.

[148] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *proceedings of the ECOOP 2005 19th European Conference Object-Oriented Programming.*, pages 504–527, Glasgow, Scotland, July 27-29, 2005.

[149] M.R. Paige. An analytical approach to software testing. In *Proc. IEEE Computer Society's Second International Computer Software and Applications Conference COMPSAC '78*, pages 527–532, 1978.

[150] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(3):263–282, Sep 1999.

[151] Moon-Won Park and Yeong-Dae Kim. A systematic procedure for setting parameters in simulated annealing algorithms. *Comput. Oper. Res.*, 25(3):207–217, 1998.

[152] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 23–32, New York, NY, USA, 2004. ACM.

[153] Hartmut Pohlheim. Genetic and evolutionary algorithms: Principles, methods and algorithms. GEATbx Tool box Matlab, version 1.91, July 1997.

[154] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions of Software Engineering*, 11(4):367– 375, April 1985.

[155] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. Wiley, December 1996.

[156] Ingo Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme und Prinzipien der biologischen Evolution*. Frommann-Holzboog, Stuttgart, 1973.

[157] R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill, 1995.

[158] Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. Derivation of domain test scenarios from activity diagrams. 2003. Proceedings of the PLEES03 International Workshop on Product Line Engineering: The Early Steps: Planning, Modeling, and Managing.

[159] Frederick G. Sayward Richard A. DeMillo, Richard J. Lipton. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, 11(4):34–41, April 1978.

[160] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *proceedings of 14th International Conference on Software Engineering*, May 1992.

[161] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. *Fundamental Approaches to Software Engineering*, LNCS 3442:282–297, 2005.

[162] RTCA/DO-178B. Software considerations in airborne systems and equipment certification. pages 2455–2464, Washington D.C., USA 1992.

[163] Johannes Ryser and Martin Glinz. Using Dependency Charts to Improve Scenario-Based Testing. In *proceedings of 17th International Conference on Testing Computer Software TCS2000*, Washington D.C, 2000.

[164] Sagarna and J.A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence.*, 19(5):457–489., 2005.

[165] Sagarna and J.A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412., 2006.

[166] Ramn Sagarna. *An Optimization Approach for Software Test Data Generation: Applications of Estimation of Distribution Algorithms and Scatter Search.* PhD thesis, University of the Basque Country, 2007.

[167] Philip Samuel, Rajib Mall, and Pratyush Kanth. Automatic test case generation from UML communication diagrams. *Inf. Softw. Technol.*, 49(2):158–171, 2007.

[168] Peter H. Schmitt and Benjamin Wei. Inferring Invariants by Symbolic Execution. In *proceedings of the 4th International Verification Workshop (VERIFY'07)*, Bremen, Germany, 2007.

[169] G. T. Scullard. Test case selection using VDM. In *proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead*, pages 178–186, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[170] S. Sendall, R. Hauser, J. Koehler, J. K
"uster, and M. Wahler. Understanding Model Transformation by Classification and Formalization. In *Software Transformation Systems Workshop*, pages 30–31, Vancouver, Canada, October 2004.

[171] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20:42–45, 2003.

[172] Harbhajan Singh, Mirko Conrad, and Sadegh Sadeghipour. Test Case Design Based on Z and the Classification-Tree Method. In *ICFEM '97: Proceedings of the 1st International Conference on Formal Engineering Methods*, page 81, Washington, DC, USA, 1997. IEEE Computer Society.

[173] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, 2004.

[174] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing'. In *proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 73 – 81, Clearwater Beach, Florida, United States, 1998.

[175] Nigel Tracey, John Clark, Keith Mander, and John A. McDermid. An automated framework for structural test-data generation. In *proceedings of the Automated Software Engineering (ASE'98)*, pages 285–288, 1998.

[176] Nigel J. Tracey. *A Search-Based Automated Test-data Generation Framework for safety-critical Softwares*. PhD thesis, The University of York, 2000.

[177] Andy Tripp. http://jazillian.com/antlr/emitter.html.

[178] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analyis using mutant schemata. In *ISSTA '93: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis*, pages 139–148, New York, NY, USA, 1993. ACM.

[179] Lionel van Aertryck, Marc Benveniste, and Daniel Le Mtayer. CASTING: A Formally Based Software Test Generation Method. *Formal Engineering Methods, International Conference on*, 0:101, 1997.

[180] William T. Vetterling and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1992.

[181] Jeffrey M Voas and Gary McGraw. *Software Fault Injection*. John Wiley and Sons Inc, 1998.

[182] K. S. How Tai Wah. An analysis of the coupling effect i: single test data. *Sci. Comput. Program.*, 48(2-3):119–161, 2003.

[183] Stefan Wappler, Andre Baresel, and Joachim Wegener. Improving Evolutionary Testing in the Presence of Function-Assigned Flags. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[184] Stefan Wappler and Frank Lammermann. Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software. In *proceedings of the 2005*

*Conference on Genetic and Evolutionary Computation (GECCO '05)*, pages 1053–1060. ACM, 2005.

[185] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932, New York, NY, USA, 2006. ACM.

[186] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841 – 854, 2001.

[187] Dennis Weyland. Simulated annealing, its parameter settings and the longest common subsequence problem. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 803–810, New York, NY, USA, 2008. ACM.

[188] M Winters. *Quality Assurance for Object-Oriented Software Requirements Engineering and Testing with respect to Requirement Specification.* PhD thesis, University of Hagen, September 1999.

[189] M.R. Woodward and K Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 152–158, Banff, Alberta, Canada, July 1988. IEEE Computer Society Press.

[190] Y. Wu, M.H. Chen, and J. Offutt. UML-based integration testing for component-based software. *Lecture Notes in Computer Science*, 2580:251–260, 2003.

[191] S Xanthakis, C Ellis, C Skourlas, A Le Gall, S Katsikas, and K Karapoulios. Application of genetic algorithms to software testing (application des algorithmes gntiques au test des logiciels. In *5th International Conference on Software Engineering and its Applications*, 1992.

[192] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 49–58, Montreal, Canada, October 8-10 2003.

[193] Xin Yao. Comparison of different neighbourhood sizes in simulated annealing. In *Proc. of Fourth Australian Conf. on Neural Networks*, pages 216–219, 1993.

[194] S.J. Zeal. Testing for perturbation of program statements. *IEEE Transaction on Software Engineering*, 9(3):335–346, May 1983.

[195] Yuan Zhan. *Search Based Test Data Generation for Simulink Models*. PhD thesis, University of York, 2005.

[196] Yuan Zhan and John A. Clark. Search Based Automatic Test-Data Generation at an Architectural Level. In *proceedings of the 2004 Conference on Genetic and Evolutionary Computation (GECCO'04)*, pages 1413–1424. Springer, 2004.

[197] Yuan Zhan and John A. Clark. Search-based mutation testing for Simulink models. In *proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*, pages 1061–1068. ACM, 2005.

[198] Yuan Zhan and John A. Clark. The state problem for test generation in Simulink. In *proceedings of the 8th annual Conference on Genetic and Evolutionary Computation (GECCO '06)*, pages 1941–1948. ACM, 2006.

[199] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.