Improving Fault Coverage and Minimising the Cost of Fault Identification when Testing from Finite State Machines

A thesis submitted in fulfilment of the requirement for
the degree of Doctorate of Philosophy

Qiang Guo

School of Information Systems, Computing and Mathematics

Brunel University
Uxbridge, Middlesex
UB8 3PH

United Kingdom

January 9, 2006

To my parents,

and to Yuyan ...

Acknowledgements

I would like to thank Professor Robert M. Hierons of Brunel University, who is my first supervisor, and Professor Mark Harman of King's College London, who is my second supervisor, for their excellent supervision and support throughout this research.

My thank also goes to people from VASTT (Verification and Analysis using Slicing, Testing and Transformation) and ATeSST (The Analysis, Testing, Slicing, Search and Transformation) research groups in Brunel University, and the EPSRC funded FORTEST and SEMINAL networks for their valuable comments and advice.

This work is funded by Brunel Research Initiative and Enterprise Fund (BRIEF) Award from Brunel University for the first year, and then departmental bursary from school of information systems, computing and mathematics, Brunel University, for the second and third years.

Related publications

- Qiang Guo, Robert M. Hierons, Mark Harman and Karnig Derderian, “Computing unique input/output sequences using genetic algorithms”, *Formal Approaches to Testing (FATES’03)*, in LNCS 2931:164-177, 2004.
- Qiang Guo, Robert M. Hierons, Mark Harman and Karnig Derderian, “Constructing multiple unique input/output sequences using metaheuristic optimisation techniques”, *IEE Proceedings - Software*, 152(3):127-130, 2005.
- Qiang Guo, Robert M. Hierons, Mark Harman and Karnig Derderian, “Improving test quality using robust unique input/output circuit sequences (UIOCs)”, *Information and Software Technology*, accepted for publication, 2005.
- Qiang Guo, Robert M. Hierons, Mark Harman and Karnig Derderian, “Heuristics for fault diagnosis when testing from finite state machines”, *Software Testing, Verification and Reliability*, under review, 2005.

Abstract

Software needs to be adequately tested in order to increase the confidence that the system being developed is reliable. However, testing is a complicated and expensive process. Formal specification based models such as finite state machines have been widely used in system modelling and testing. In this PhD thesis, we primarily investigate fault detection and identification when testing from finite state machines.

The research in this thesis is mainly comprised of three topics - construction of multiple Unique Input/Output (UIO) sequences using Metaheuristic Optimisation Techniques (MOTs), the improved fault coverage by using robust Unique Input/Output Circuit (UIOC) sequences, and fault diagnosis when testing from finite state machines. In the studies of the construction of UIOs, a model is proposed where a fitness function is defined to guide the search for input sequences that are potentially UIOs. In the studies of the improved fault coverage, a new type of UIOCs is defined. Based upon the Rural Chinese Postman Algorithm (RCPA), a new approach is proposed for the construction of more robust test sequences. In the studies of fault diagnosis, heuristics are defined that attempt to lead to failures being observed in some shorter test sequences, which helps to reduce the cost of fault isolation and identification. The proposed approaches and techniques were evaluated with regard to a set of case studies, which provides experimental evidence for their efficacy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 About testing

Development of software systems is comprised of three stages. In the first stage, developers of the system derive a set of *requirements* from their customers. These requirements are normally represented in a requirements *specification*. Then, in consultation with these requirements, a *design* is built. After that, *Coding*, or *implementing* takes place where the design is translated into code using some programming language. Errors might be introduced at this stage, but can be discovered by verification and testing. Testing is an integral and important part in the life cycle of software development. A testing process aims to check whether the implementation under test is functionally equivalent to its specification.

"*Testing is the process of executing a program or system with the intent of finding errors, or, involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results*" (Het88; Mye79).

The process of testing begins with test design. A set of tests is normally created through the analysis of the system under test. This set of tests is used to check whether the system has been correctly implemented. In the phase of test design, a test model is often required in order that the generation of tests is formalised. This model describes the system behaviour with abstracted information, aiming to reduce the complexity of the description of the system being developed. The test model can be constructed by using either informal specification languages or formal specification languages. Due to the properties of imprecision and ambiguity, informal specifications often lead to misunderstandings and make testing difficult and unreliable. By contrast, formal specification languages are based upon mathematics and have a formally defined semantics. The mathematical nature of formal specification languages leads to precise and unambiguous descriptions. Section 1.3 gives a brief review on the major formal specification languages.

After a test model is built, a test strategy needs to be defined for the generation of test cases. A test strategy is an algorithm or heuristic to create test cases. Two measurements are applied for the evaluation of efficiency of a test. One of the measurements is *test cost* while the other is *fault coverage*. A good test strategy needs to embody the two measurements in two aspects: (1) test cases generated with such a strategy should cover, as much as possible, all faults that the system under test may have; (2) test cost associated with these test cases should be relatively low.

When design is complete, a set of test cases is then applied to the system under test to check its correctness. With the input set of a test case being applied to the system, an output set will be received. The application of a test case is classified as *pass* or *fail* by comparing the output set to that defined in the specification. Failure caused by any test case suggests the existence of faults in the system under test.

The procedure of testing is summarised as four major steps:

1. Identify, model, and analyse the responsibilities of the system under test.

2. Design test cases based on this external perspective.

3. Develop expected results for each test case or choose an approach to evaluate the pass/fail status of each test case.

4. Apply test cases to the system under test.

Unfortunately, detecting all faults is generally infeasible. Howden (How76) suggests that there is no algorithm to find consistent, reliable, valid, and complete test criteria. Complete testing is in general a very difficult process. Instead, testing provides a level of confidence in the correctness of an implementation with regard to the constraint of some test criteria. Exhaustive testing, where the test cases consist of every possible set of input values, is the only way that will guarantee complete fault coverage. This technique, however, is not practical. The size of the input domain makes exhaustive testing infeasible (And86).

Regardless of the limitations, testing is an expensive process, typically consuming at least 50 % of the total costs involved in the development (Bei90) while adding nothing to the functionality of the product. It has been suggested that manually generating test cases could be very difficult even for moderately sized systems (Mye79). Although, for some systems, it is possible to generate test cases manually, the process tends to be costly and inefficient. Automation of the testing process is thus required, which could be desirable both to reduce development costs and to improve the quality of (or at least confidence in) software.

1.2 Validation and verification

Validation and *verification* are essential in the life cycle of system development. Without rigorous validation, verification and testing that the specification meets the customer's requirements and that the implementation is consistent with its specification, the development of a system is not complete.

1.2.1 Validation

Validation is defined as "*the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements*" (IEE90). Validation checks that the system being developed conforms to the user's requirements. It generally answers the question, "*Did we build the right system?*" (Boe81).

Early validation of the system specification is very important. Before being translated into an implementation, a specification needs to be checked for validity, consistency, competence, realism and verifiability.

The process of validation is classified into two stages - *informal validation* and *formal validation*. In system development, once the writing of specification and the coding of implementation are complete, a set of validation tests needs to be developed. The set of tests aims to report the majority of problems in the system being developed. This activity is referred to as informal validation since tests are run informally, and some system features are expected to be missing. Informal validation provides early feedback to software engineers. This feedback provide the system developers with foundations for system modifications, which help to increase confidence that the system being developed complies with the customer's requirements.

After informal validation is complete, the process comes to formal validation. A set of tests is designed and applied to the implementation under test with the purpose of bug correction. At this stage, the specification used for system coding is assumed to be valid and complete. The process of testing aims to check whether the implementation under test conforms to the specification. A formal test model is often defined. Formal approaches are applied for the derivation of test cases.

Validation is usually accomplished by verifying each stage of the software development life cycle. It should be noted that, in reality, specification validation is normally unlikely to discover all requirements problems. Some flaws and deficiencies in the specification can sometimes only be discovered when the system implementation is complete.

1.2.2 Verification

Verification is defined as "*the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase*" (IEE90). Verification involves checking that the implementation produced conforms to the specification. It addresses the question, "*Did we build the system right?*" (Boe81). Many techniques have been proposed for the verification but they all basically fall into two major categories: static verification and dynamic verification.

I. Static verification

Static verification is concerned with analysing the system being developed without executing it. Properties of the system such as syntax, parameter matching between procedures, typing and specification translation have to be checked for their correctness.

Software inspection (AFE84) is one of the static verification techniques that has been widely used. "*A software inspection is a group review process that is used to detect and correct defects in a software work-product. It is a formal, technical activity that is performed by the work-product author and a small peer group on a limited amount of material. It produces a formal, quantified report on the resources expended and the results achieved*" (AFE84).

During inspection, either the code or the design of a work-product is compared to a set of pre-established inspection rules. Inspection processes are mostly performed along checklists which cover typical aspects of the software behaviour.

Another static verification technique is walk-through (Tha94). Walk-throughs are similar peer review processes that involve the author of the program, the tester and a moderator. The participants of a walk-through create a small number of test cases by simulating the computer. Its objective is to question the logic and basic assumptions behind the source code, particularly of program interfaces in embedded systems (Tha94).

Proofs are usually used for the verification, either informally or formally. At an informal level, proofs work on step-by-step reasoning involved in an inspection of the system while, at a more formal level, mathematical logic is introduced. A formal proof is based upon a set of axioms and inference rules. If the test model is constructed with a formal language, a set of formal proofs can be derived to prove the conformance of implementation to its specification. As formal proofs can be checked automatically in a formal language, an automatic proof checker can be used for the construction of proofs.

However, it is usually difficult to prove the conformance to the specification for a large scale system. This can be alleviated through the use of refinement. Specification, in some formal notation, can be converted into an implementation using a series of simple refinements, each of which is capable of being proven. By such an operation, no fault should be present in the implementation.

II. Dynamic verification

Dynamic verification involves the execution of a system or component. A number of test inputs are chosen. Corresponding test outputs are used to determine the gap between the test model and the real implementation. These test inputs are called *test cases* and the process is called *testing*.

Testing consists of three major stages, namely, *unit* or *module testing*, *integration testing* and *system testing*. In the stage of module testing, modules are tested individually, aiming to find defects in logic, data, and algorithms. In the stage of integration testing, modules are grouped with regard to their functionalities, each group being tested as an integrated whole. Once integration testing has finished, testing comes to the stage of system testing where the implementation is thoroughly tested as a system, taking more comprehensive factors into account.

Testing can be further divided into three categories - functional testing, structural testing, and random testing.

A: Functional testing

Functional testing aims to identify and test all functions of the system defined in the specification. Involving no knowledge of the implementation of the system, functional testing is a type of black-box testing. Category partition (OB89) is the most widely used technique in functional testing. It involves five steps:

1. Analyse the specification to identify individual functional units;
2. Identify parameters and environment variables of the functional unit;
3. Identify the categories for each parameter and environment variable;
4. Partition each category into a set of choices and possible values;
5. Specify the possible results and the changes to the environment.

Two advantages can be noticed in category partition method. First, the test set is derived from the specification and therefore has a better chance of detecting whether some functionalities are missed from the implementation. Second, the test phase can be started early in the development process and the test set can be easily modified as the system evolves.

However, it is difficult to formally define categories and choices. This could make it very hard to assess whether the criteria used for partition are adequate. As a result, the generation of partitions relies heavily on the experience of testers.

B: Structural testing

Structural testing is a type of white-box testing. It uses the information from the internal structure of the system to devise tests to check the operation of individual components. Three scopes are addressed in structural testing - *Statement Coverage*, *Branch Coverage* and *Path Coverage*. If, in a test, the test set causes every statement of the code to be executed at least once, then statement coverage is achieved, while, if the test set causes every branch to be executed at least once, then branch coverage is achieved. In other words, for every branch statement, each of the possibilities must be performed on at least one occasion. If the test set causes every distinct execution path be taken at some point, then path coverage is achieved.

C: Random testing

Random testing randomly chooses test cases from the test domain. It provides a means to detect faults that remain undetected by the systematic methods. Exhaustive testing where the test cases consist of every possible set of input values is a form of random testing. Although exhaustive testing guarantees a complete fault coverage for the system being developed, it is impossible to accomplish in practice (And86).

1.2.3 Validation vs. verification

Validation and verification are highly related to software quality. With the increasing complexity of systems, validation and verification become more and more important. Without validation, an incomplete specification might be acquired, leading to an inadequate design and an incorrect implementation; while, without verification, no proof is exhibited that an implementation conforms to its specification. Planning for validation and verification is often viewed as a very important step from the beginning of the development.

Validation and verification can be conducted in parallel within a project as they are not mutually exclusive.

1.3 Formal specification languages

Writing specification from customer requirements is a key activity in the development of systems. A well-defined requirement specification language is considered to be a prerequisite for efficient and effective communication between the users, requirements engineer and the designer. Specification languages provide frames where problems are defined and solved. They provide operators that are used in analysing, manipulating and transforming the system description.

Requirements specification languages may be classified into two major classes: informal specification languages and formal specification languages. Formal specification language have a mathematical (usually formal logic) basis and employ a formal notation to model system requirements (AG88) while informal specification languages use a combination of graphics and semiformal textual grammars to describe and specify system requirements. Despite some 'formalising' efforts at the specification and design, informal specifications tend to be ambiguous and imprecise, which might lead to misunderstanding and makes it difficult to detect inconsistencies and incompleteness in the specification.

By contrast, by using the formal notation, precision and conciseness of specifications can be achieved. As a formal notation can be analysed and manipulated using mathematical operators, mathematical proof procedures can be used to test (and prove) the internal consistency and syntactic correctness of the specifications. In addition, by using formal notation, the completeness of the specification can be checked in the sense that all enumerated options and elements have been specified.

Three main types of formal specification languages have been proposed for the system description, these being:

1. Model oriented specification languages

2. Algebraic specification languages

3. Process algebras

Model oriented specification languages

Model oriented specification languages are aimed to build up a mathematical model for the system being developed. The specification is written with a model oriented language where objects such as data structures and functions are mathematically described in details. These mathematical objects are structurally similar to the system required. During the design and implementation, mathematical objects are transformed in ways that preserve the essential features of the requirements as initially specified.

It is characteristic of model oriented languages that the model of the system is given by describing the state of the system, together with a number of operations over that state. An operation is a function which maps a value of the state together with values of parameters to the operation onto a new state value.

The most widely known model oriented specification languages are VDM-SL, the specification language associated with VDM (Jon90), the Z specification language (Spi88; Spi89) and the B specification language (Abr96).

Algebraic specification languages

Algebraic specification languages such as OBJ3 (GW88) specify information systems using methods derived from abstract algebra or category theory. Abstract algebra is the mathematical study of certain kinds or aspects of structure abstracted away from other features of the objects under study. Algebraic methods are beneficial in permitting key features of information systems to be described without prejudicing questions that are intended to be settled later in the development process (implementation detail).

Process algebras

Process algebras are best described as a set of formalisms for modelling systems that allow for mathematical reasoning with respect to a set of desired properties, be it equivalence, absence of deadlocking or some safety properties. Process algebras involve defining a set of agents and the manner in which these agents interact, and thus are good at modelling situations in which there are a number of entities that interact by communicating with each other. By expressing concurrency, process algebras allow the analysis of this concurrency.

It is usually the case that process algebras are used for model concurrent systems and communication systems. The best known process modelling languages are CSP (Hoa85), CCS (Mil89) and LOTOS (fSI88).

Finite state machines (Koh78) are a less general type of process algebra. In chapter 4, testing from finite state machines is discussed. However, the use of finite state machines has disadvantages where they are not able to express non-determinism and concurrency either as elegantly or as powerfully as the more general process algebras.

The formal languages above look at systems in different ways and, consequently, represent information in different forms. The selection of a type of formal specification language for modelling a system depends upon the nature of the system being developed.

It should be noted that the use of formal specification languages might also lead to some disadvantages. One major issue is that requirements usually change during a project, which makes the procedure of determining a final specification expensive. It was suggested that it is very expensive to develop a formal specification of a system, and it is even more expensive to show that a program meets that specification (AG88).

1.4 Test cost and fault coverage

Two factors, *test cost* and *fault coverage*, are tightly coupled with the evaluation of a test. Test cost involves the numbers of test data that are used for the verification of the system under test while fault coverage considers the percentage of faults that have been detected by such a test. It is always desirable that a test will achieve complete fault coverage with the lowest test cost.

Test cost and fault coverage, however, sometimes counteract one another. On one hand, a system needs to be tested with enough test data in order that the complete fault coverage has been achieved. The more test data are applied, the more deficiencies will be detected. Exhaustive testing guarantees the complete fault coverage for the system under test. However, tests that guarantee complete fault coverage are sometimes too long for practical applications, which will consequently result in a higher test cost. Tests using less test data are always preferred; on the other hand, too little test data might cause some deficiencies to be missed by the test, leading to an incomplete test. The problem of test cost leads to the study of test optimisation while the problem of fault coverage leads to the study of test quality.

An effective test often requires a trade-off to be made between the test cost and the fault coverage. A good test generation strategy needs to compromise between the two factors in two aspects: (1) test cases generated with such a strategy should cover, as much as possible, all faults that the system under test may have; (2) the test cost associated with such test cases should be comparatively low.

Optimisation on test cost with regard to fault coverage has been thoroughly studied when finite state machines are applied (ATLU91; Hie97; MP93; SLD92; YU90). In chapter 4, testing from finite state machines is discussed. This PhD work has investigated the problem of test quality when testing from finite state machines. In the work, robust Unique Input/Output Circuit (UIOC) sequences were defined for state verifications. Based on rural Chinese postman algorithm, a new test generation algorithm is given. Experimental results suggest that the proposed method leads to a more robust test sequence than those constructed with the existing methods without significantly increasing the test length. The work is discussed in chapter 6.

1.5 Fault observation and diagnosis

An important yet complicated issue associated with testing is fault diagnosis. The process of testing aims to construct test cases that could be used to provide confidence that the implementation under test conforms to its specification.

Usually, a system is modelled as a set of functional units (components), some of which are connected with others through input and output coupling. Each unit is assigned with two attributes: an *I/O port* and an *internal state*. I/O port provides testers with an interface for the observation of outputs when inputs are sent, while, the internal state is not visible and can only be inferred through exhibited input/output behaviour. Once a test case is constructed, it is applied to an implementation, all units being executed successively. I/O differences exhibited between the implementation and the specification suggest the existence of faults in the implementation. The first observed faulty I/O pair in an observed I/O sequence is called a *symptom*. A symptom could have been caused by either an incorrect output (an *output fault*) exhibited by the unit being tested, or an earlier incorrect state transfer (a *state transfer fault*) that remains unexhibited in the units that have already been executed by the checking data. It is therefore important to define strategies to guide the construction of test data. These data could be used to (effectively) isolate the faulty units in the implementation that will explain the symptoms exhibited.

The process of isolating faults from the implementation with regard to the symptoms observed is called *fault diagnosis* (LY96).

However, fault diagnosis is very difficult. Very little work has been done for the diagnostic and the fault localisation problems (GB92; GBD93). Steinder and Sethi (SS04) proposed a probabilistic even-driven fault propagation model where a probabilistic symptom-fault map is used for the process of fault diagnosis. The technique utilises a set of hypotheses that most probably explains the symptoms observed at each stage of evaluation. The set of hypotheses is updated with the process going further, maximising the probabilities of hypotheses for the explanation of observed symptoms.

Ghedamsi and Bochmann (GB92; GBD93) modelled the process of fault diagnosis with finite state machines. A set of transitions is generated whose failure could explain the behaviour exhibited. These transitions are called *candidates*. They then produce tests (called distinguishing tests) in order to find the faulty transitions within this set. However, in the approach, the cost of generating a conflict set is not considered.

Hierons (Hie98) extended the approach to a special case where a state identification process is known to be correct. Test cost is then analysed by applying statistical methods. Since the problem of optimising the cost of testing leads to

NP-hard (Hie98), heuristic optimisation techniques such as *Genetic Algorithms* and *Simulated Annealing* are suggested.

This PhD work studied the problem of fault diagnosis. In the work, heuristics are defined for fault isolation and identification when testing from finite state machines. The proposed approach attempts to lead to a symptom being observed in some shorter test sequences, which helps to reduce the cost of fault isolation and identification. The work is discussed in chapter 7.

1.6 Testing with MOTs

Metaheuristics Optimisation Techniques (MOTs) such as Genetic Algorithms (GAs) (Gol89) and Simulated Annealing (SA) (KGJV83) are widely used in the problems of search and optimisation. More recently, MOTs have been successfully applied in software engineering, including automating the generation of test data. Examples of such applications can be found in structural coverage testing (branch coverage testing) (JES98; MMS01), worst case and best case execution time estimation (WSJE97), and exception detection (TCMM00).

MOTs are search techniques that simulate nature. When using MOTs, an objective function is defined to guide the search of solutions for the problem under investigation. The objective function is called the *fitness function*. The search process could be aimed at either maximising or minimising the fitness function. An iteration scale is defined to determine the computational times. At each step of the computation, a new solution is provided. By evaluating its fitness value, the solution will be either accepted or rejected. In chapter 2, some MOTs are introduced.

Automating the generation of test data is of great value in reducing the development cost and improving the quality in software development. MOTs provide means to automate such a process. The reasons that MOTs are used in the generation of test cases are: (1) the problem of generating test cases is equivalent to a search problem where good solutions need to be explored in the input space of the system being developed. Usually, the input space is large. This could make the search a costly and inefficient process when traditional algorithms are applied. This problem, however, can be alleviated by heuristic search, such as MOTs;

(2) some problems in testing such as the construction of unique input/output sequences are NP-hard problems and MOTs have proved to be efficient in providing good solutions for NP-hard problems.

This PhD work investigated the construction of multiple Unique Input/Output (UIO) sequences by using MOTs. In the work, a fitness function is defined to guide the search of input sequences that constitute UIOs for some states. The fitness function works by encouraging the early occurrence of discrete partitions in the state splitting tree constructed by an input sequence while punishing the length of this input sequence. The work and the experimental results are discussed in chapter 5.

1.7 The structure of this thesis

This thesis is comprised of eight chapters. It is organised as follows: chapter 1 briefly introduces the background of testing; chapter 2 defines the preliminaries and notation used in this thesis; chapter 3 reviews the major test generation techniques; chapter 4 reviews the automated generation of test cases when testing from finite state machines; chapter 5 studies the construction of Unique Input/Output (UIO) sequences and proposes a model for the construction of multiple UIOs using Metaheuristic Optimisation Techniques (MOTs); chapter 6 investigates the fault coverage in finite state machine based testing and proposes a new type of Unique Input/Output Circuit (UIOC) sequence for state verification. Based upon Rural Chinese Postman Algorithm (RCPA), a new approach is proposed for the generation of test sequences from the finite state machine under test; chapter 7 looks at fault diagnosis when testing from finite state machines, and proposes heuristics for fault isolation and identification; in chapter 8, conclusions are drawn. Some future work is also suggested in chapter 8.

Chapter 2

Preliminaries and notation

2.1 Graph theory

The automated generation of test cases benefits from the applications of graph theory when testing from finite state machines. In this section, preliminaries and notation of graph theory are introduced. Terminologies, notation and algorithms are mainly cited from ref. (BJG01).

2.1.1 Directed graph

Definition 2.1.1 *A* graph *G is a pair (V, E) where V is a set of vertices, and E is a set of edges between the vertices $E \subseteq \{\{u, v\}|u, v \in V\}$.*

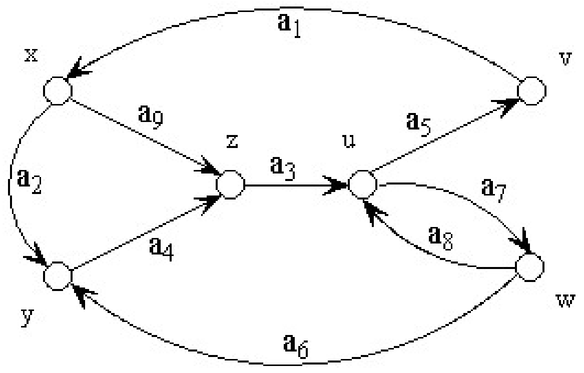


Figure 2.1: An example of labelled digraph.

Definition 2.1.2 *A* labelled digraph $G = (V, E, \Sigma)$ *is a directed graph with vertex set V, label set Σ and edge function E: $V \times \Sigma \to V$, $E(u, \sigma) = v$ where $u, v \in V$ and $\sigma \in \Sigma$.*

An example of a labelled digraph is illustrated in Figure 2.1 where $V(D) = \{u, v, w, x, y, z\}$, $E(D) = \{(u, v; a_5)$, $(u, w; a_7)$, $(w, u; a_8)$, $(z, u; a_3)$, $(x, z; a_9)$, $(y, z; a_4)$, $(v, x; a_1)$, $(x, y; a_2)$, $(w, y; a_6)\}$ and $\Sigma = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$. A labelled digraph is a special case of digraph where each edge is labelled with characters, indicating the relation between two vertices of the edge.

The number of vertices in a digraph D is called the *order* or *size*. An edge $(u, v) \in E(D)$ leaves u and enters v. u is the *head* of the edge and v the *tail*. The head and tail of an edge are its *end-vertices*; the end-vertices are *adjacent*, i.e. u is adjacent to v and v is adjacent to u. For a vertex $v_i \in V$, the *in-degree*, $d^+(v_i)$, is the number of inward edges to v_i; the *out-degree*, $d^-(v_i)$, is the number of outward edges from v_i. The index of a vertex $\xi(v_i)$ is defined as the difference between the out-degree and in-degree of this vertex, $\xi(v_i) = d^-(v_i)$ - $d^+(v_i)$. For example, the order of the labelled digraph shown in Figure 2.1 is 6; in the digraph, edge (x, y) leaves vertex x and enters y; x is the head of (x, y) and y the tail; $d^+(x) = 1$ while $d^-(x) = 2$; $\xi(x) = 1$.

Definition 2.1.3 *A digraph D is* symmetric *if, for every vertex $v_i \in V$, $d^+(v_i) = d^-(v_i)$.*

Definition 2.1.4 *A* walk *in D is an alternating sequence $W = v_1e_1v_2e_2v_3 \ldots v_{k-1}e_{k-1}v_k$ of vertices $v_i \in V(D)$ and edge $e_i \in E(D)$ such that the head of e_i is v_i and the tail of e_i is v_{i+1} for every $i = 1, 2, ..., k-1$. The* length *of a walk is the number of its edges.*

The set of vertices $\{v_1, v_2, ..., v_k\}$ in a walk W is denoted by $V(W)$ and the set of edges $\{e_1, e_2, ..., e_{k-1}\}$ is denoted by $E(W)$. W is a walk from v_1 to v_k or an *(v_1, v_k)-walk*. A walk W is *closed* if $v_1 = v_k$ and *open* otherwise. If $v_1 \neq v_k$, then the vertex v_1 is the *initial* vertex of W, the vertex v_k is the *terminal* vertex of W, and v_1 and v_k are *end-vertices* of W. A walk W is a *trail* if all edges in W are distinct; a vertex v_i is *reachable* from a vertex v_j if D has an (v_i, v_j)-walk. $W_1 =$

$v(v,x)x(x,y)y(y,z)z(z,u)u(u,w)w$ and $W_2 = v(v,x)x(x,y)y(y,z)z(z,u)u(u,v)v$ are two walks in Figure 2.1 where W_1 is open while W_2 closed. Edges in both walks are distinct and, therefore, both walks are trails; vertex w is reachable from vertex v. v is the initial vertex of W_1 while w is the terminal.

Definition 2.1.5 *A walk W is a* path *if the vertices of W are distinct; W is a* cycle *if the vertices v_1, v_2, ..., v_{k-1} are distinct, $k \geq 3$ and $v_1 = v_k$.*

W_1 shown above is a path while W_2 is a cycle. A path P is an *$[v_i, v_j]$-path* if P is a path between v_i and v_j, e.g. P is either an *(v_i, v_j)-path* or an *(v_j, v_i)-path*. An (v_i, v_j)-path $P = v_1v_2...v_n$ is *minimal* if, for every (v_i, v_j)-path Q, either $V(P) = V(Q)$ or Q has a vertex not in $V(P)$.

Definition 2.1.6 *A* tour *is a walk that starts and ends at the same vertex. An* Euler tour *in a digraph D is a tour that contains every edge of $E(D)$ exactly once. A* postman tour *of a digraph D is a tour that contains every edge of $E(D)$ at least once. A* Chinese postman tour *is a postman tour where the number of edges contained in the tour is minimal.*

It is easy to see that an Euler tour is also a Chinese postman tour.

Definition 2.1.7 *A digraph D is* strongly connected *(or, simply,* strong*) if, for every pair v_i, v_j of distinct vertices in D, there exists a (v_i, v_j)-walk and a (v_j, v_i)-walk. In other words, D is strongly connected if every vertex of D is reachable from every other vertex of D. A digraph D is* weakly connected *if the underlying undirected graph is connected.*

Definition 2.1.8 *A digraph H is a* subdigraph *of a digraph D if $V(H) \subseteq V(D)$, $E(H) \subseteq E(D)$ and every edge in $E(H)$ has both end-vertices in $V(H)$. H is said to be a* spanning subdigraph *(or a* factor*) of D if $V(H) = V(D)$.*

Definition 2.1.9 *The* edge-induced subgraph *$D = (V^{'}, E^{'})$ of a digraph D for some set $E^{'} \subseteq E$ is the subgraph of D whose vertex set is the set of ends of edges in $E^{'}$ and whose edge set is $E^{'}$. $D = (V^{'}, E^{'})$ is an* edge-induced spanning *subgraph of D if $V^{'} = V$.*

Lemma 2.1.1 *(Kua62) A digraph D contains an Euler tour if and only if D is strongly connected and symmetric.*

Lemma 2.1.2 *(EJ73) An Euler tour of a symmetric and strongly connected digraph D can be computed in linear time $O(n)$ where n is the number of edges in D.*

2.1.2 Flows in networks

Definition 2.1.10 *A* network $\mathcal{N} = (V, E, l, u, b, c)$ *is a directed graph $D = (V, E)$ associated with the following functions on $V \times V$: a* lower bound $l_{ij} \geq 0$, *a* capacity $u_{ij} \geq l_{ij}$, *a* cost c_{ij} *for each $(i, j) \in V \times V$ and a* balance vector $b : V \to \mathbb{R}$ *that associates a real number with each vertex of D. These parameters satisfy the condition that for every $(i, j) \in V \times V$, if $(i, j) \notin E$, then $l_{ij} = u_{ij} = 0$.*

Definition 2.1.11 *A* flow x *in a network $\mathcal{N}$ is a function $x : E \to \mathbb{R}$ on the edge set of $\mathcal{N}$; the value of x on the edge (i, j) is denoted as x_{ij}. An* integer flow *in $\mathcal{N}$ is a flow x such that $x_{ij} \in \mathbb{Z}$ for every edge (i, j).*

For a given flow x in $\mathcal{N}$ the *balance vector* of x is the following function b_x on the vertices:

$$b_x = \sum_{vw \in E} x_{vw} - \sum_{uv \in E} x_{uv} \qquad \forall v \in V. \tag{2.1}$$

A vertex v is a *source* if $b_x(v) > 0$, a *sink* if $b_x(v) < 0$, and otherwise v is *balanced* ($b_x(v) = 0$). A flow x in $\mathcal{N} = (V, E, l, u, b, c)$ is *feasible* if $l_{ij} \leq x_{ij} \leq u_{ij}$ for all $(i, j) \in E$ and $b_x(v) = b(v)$ for all $v \in V$. A *circulation* is a flow x with $b_x(v) = 0$ for all $v \in V$.

The *cost* of a flow x in $\mathcal{N} = (V, E, l, u, c)$ is given by

$$c^T x = \sum_{ij \in E} c_{ij} x_{ij}. \tag{2.2}$$

where c_{ij} is the cost of edge x_{ij}.

The notation of (s, t)-paths in a digraph D can be generalised as that of flows. If P is an (s, t)-path in a digraph $D = (V, E)$, then an (s, t)-flow x can be described in the network $\mathcal{N}(V, E, l \equiv 0, u, c)$ by taking $x_{ij} = k, k \in \mathbb{Z}^+$ if (i, j) is an edge of P and $x_{ij} = 0$ otherwise. This flow has balance vector:

$$b_x(v) = \begin{cases} k, & \text{if } v = s \\ -k, & \text{if } v = t \\ 0, & \text{otherwise} \end{cases}$$

The value of an (s, t)-flow x is defined by

$$|x| = b_x(s) \tag{2.3}$$

A *path flow* $f(P)$ along a path P in $\mathcal{N}$ is a flow with the property that there is some number $k \in \mathbb{Z}^+$ such that $f(P)_{ij} = k$ if (i, j) is an edge of P and otherwise $f(P)_{ij} = 0$; a *cycle flow* is defined as flow $f(C)$ for any cycle C in D. The *edge sum* of two flows $x, x^{'}$, denoted $x, x^{'}$, is simply the flow obtained by adding the two edge flows edge-wise. Two path flow x, $x^{'}$ of the same trail can be merged into a new flow $x^{''} = x \oplus x^{'}$ as long as the edge sum of each edge does not exceed its capacity. $\oplus$ indicates that x and $x^{'}$ are decompositions of $x^{''}$.

Theorem 2.1.1 *(BJG01) Every flow x in $\mathcal{N}$ can be represented as the edge sum of some path and cycle flows $f(P_1)$, $f(P_2)$,..., $f(P_\alpha)$, $f(C_1)$, $f(C_2)$, ..., $f(C_\beta)$ with the following two properties:*

1. *Every directed path P_i, $1 \leq i \leq \alpha$ with positive flow connects a source vertex to a sink vertex.*

2. *$\alpha + \beta \leq n + m$ and $\beta < m$.*

n is the number of vertices and m the number of edges in the network.

Lemma 2.1.3 *(BJG01) Given an arbitrary flow x in $\mathcal{N}$, one can find a decomposition of x into at most $n + m$ path and cycle flows, at most m of which are cycle flows, in time $O(nm)$.*

Theorem 2.1.1 and Lemma 2.1.3 indicate that a flow x in a network $\mathcal{N}$ can be decomposed into a number of path flows in polynomial time. This provides foundation for maximising the flow in $\mathcal{N}$. If a flow in $\mathcal{N}$ can be decomposed into two sets of path flows where one is maximised (some edges in the path flow are saturated) and the other, in terms of capacities, has allowances, one can then augment flows along unsaturated paths. Once no augmentation flow is found in $\mathcal{N}$, the flow obtained is maximal. The problem of the maximum of flows is discussed in the next section.

Definition 2.1.12 *For a given flow x in network $\mathcal{N} = (V, E, l, u, b, c)$, the* residual capacity *r_{ij} from v_i to v_j is defined as:*

$$r_{ij} = (u_{ij} - x_{ij}). \tag{2.4}$$

The *residual network* with respect to flow x is defined as $\mathcal{N}_r = (V, E_{(x)}, l \equiv 0, r, c)$ where $E(x) = \{(i, j) : r_{ij} > 0\}$. A *residual edge* is an edge with positive capacity. A *residual path (cycle)* is a path (cycle) consisting entirely of residual edges.

2.1.3 The maximum flow and minimum cost problems

Two issues that are highly coupled with flows in a network are the maximum flow problem and the minimum cost problem. In this section, these two issues are introduced separately.

A. The maximum flow problem

The study of (s, t)-flows in a network $\mathcal{N}$ considers a special type of network $\mathcal{N} = (V, E, l \equiv 0, u)$ where $s, t \in V$ are special vertices that satisfy $b_x(s) = -b_x(t)$ and $b_x(v) = 0$ for all other vertices. s is called the *source* and t the *sink* of $\mathcal{N}$. An edge $(i, j) \in \mathcal{N}$ is called *saturated* if $x_{ij} = u_{ij}$. As theorem 2.1.1 states, every (s, t)-flow x can be decomposed into a number of path flows along (s, t)-paths and some cycle flows[1], each flow of such paths being a path flow. x is also said to be a flow *from s to t*. Its value $|x|$ is denoted by $|x| = b_x(s)$. An (s, t)-flow of value k in a network $\mathcal{N}$ is called a *maximum flow* if k is of the maximum value. The problem of finding a maximum flow from s to t is known as *maximum flow problem*. It is easy to see that an (s, t)-flow in a network $\mathcal{N}$ is *maximal* if every (s, t)-path in $\mathcal{N}$ uses at least one saturated edge $(i, j) \in \mathcal{N}$.

[1]It should be noted that the values of these cycle flows do not affect the value of the flow x.

Let x be an (s,t)-flow in $\mathcal{N}$ and P be an (s,t)-path such that $r_{ij} \geq \epsilon > 0$ for each edge (i,j) on P. Let $x^{''}$ be an (s,t)-path flow of value ϵ in $\mathcal{N}(x)$ that is obtained by sending ϵ units of flow along the path P. Let $x^{'}$ be a new flow that is obtained by $x^{'} = x \oplus x^{''}$. $x^{'}$ is of value $|x| + \epsilon$. P is called *an augmenting path* with respect to x. The *capacity* $\delta(P)$ of P is given by:

$$\delta(P) = min\{r_{ij} : (i,j) \in \mathcal{N}\}. \tag{2.5}$$

An edge (i,j) of P is a *forward edge* if $x_{ij} < u_{ij}$; (i,j) and a *backward edge* if $x_{ji} > 0$. It is obvious that an (s,t)-flow x is not maximal, if there exists an augmenting path for x.

Theorem 2.1.2 *(CCPS98) A flow x in $\mathcal{N}$ is a maximum flow if and only if $\mathcal{N}$ has no augmenting paths.*

Theorem 2.1.3 *(CCPS98) If all of the edge capacities in $\mathcal{N}$ are integral, then the there exists an integer maximum flow.*

The *minimum cut problem* is closely related to computing the maximum flow from a network. In the minimum cut problem, the input is the same as that of the maximum flow problem. The goal is to find a partition of the nodes that separates the source and sink so that the capacity of edges going from the source side to the sink side is minimum.

Definition 2.1.13 *An (s,t)-*cut *is a set of edges of the form $(S, \bar{S})$ where S,$\bar{S}$ form a partition of V such that $s \in S$, $t \in \bar{S}$. The* capacity *of an (s,t)-cut $(S,\bar{S})$ is the number $u(S,\bar{S})$, that is, the sum of the capacities of edges with tail in S and head in $\bar{S}$.*

Definition 2.1.14 *A* minimum (s,t)-cut *is an (s,t)-cut $(S, \bar{S})$ with $u(S, \bar{S}) = min\{u(S^{'}, \bar{S}^{'})$:$(S^{'}, \bar{S}^{'})$ is an (s,t)-cut in $\mathcal{N}\}$.*

It can be noted that the value of any flow is less than or equal to the capacity of any (s,t)-cut. Any flow sent from s to t must pass through every (s,t) cut, since the cut disconnects s from t. As flow is conserved, the value of the flow is limited by the capacity of the cut. This leads to Ford and Fulkerson's max-flow/min-cut theorem (FFF62).

Theorem 2.1.4 *(FFF62) The maximum value of any flow from the source s to the sink t in a capacitated network is equal to the minimum capacity among all (s,t)-cuts.*

It is easy to prove that Theorem 2.1.2 and Theorem 2.1.4 are equivalent. Theorem 2.1.2 and 2.1.4 motivate the augmenting path algorithm of Ford and Fulkerson's (FFF62) where flow is repeatedly sent along augmenting paths. This process terminates when no such paths remain. If the original capacities in the network are integral, then the algorithm always augments integral amounts of flow. This operation is initiated by Theorem 2.1.3. Ford and Fulkerson's augmenting path algorithm was modified by Edmonds and Karp (EK72) where the shortest paths (by considering the number of edges) are always preferred for augmentation. Edmonds and Karp proved that the algorithm has complexity $O(nm^2)$ where n is the number of vertices and m the number of edges.

B. The minimum cost flow problem

Given a network $\mathcal{N} = (V, E, l, u, b, c)$, a problem is to find a feasible flow x whose value of the cost is minimal. This problem is known as the *minimum cost flow problem*. As stated before, the cost of a flow x is given by $C(x) = \sum_{ij \in E} x_{ij} c_{ij}$. The goal of the problem is thus to find a feasible flow x where $C(x)$ is minimised.

Residual network can be used to check if a given flow x in $\mathcal{N}$ has minimum cost among all flows with the same balance vector. Let W be a cycle in $\mathcal{N}$ and it has the cost $c(W) < 0$. Let δ be the minimum residual capacity of an edge on W. Let $x^{'}$ be the cycle flow in $\mathcal{N}$ that sends δ units around W. If such a cycle flow exists in $\mathcal{N}$, a new flow $x^{''}$ can then be constructed by $x \otimes x^{'}$. The cost of $x^{''}$ is $c^T x + c^T x^{'} = c^T + \delta c(W) < c^T$ (since $c(W) < 0$). The cost of x is therefore not minimal.

Theorem 2.1.5 *(BJG01) A flow x in $\mathcal{N}$ is a minimum cost flow if and only if $\mathcal{N}$ contains no negative cost residual cycles.*

2.1.4 The Chinese postman tour

A problem in digraph theory intends to find a postman tour T in a directed and strongly connected digraph D where the sum of numbers of edges contained in T is minimal. This problem is known as the *Chinese postman problem* (Kua62), and such a tour is called a *Chinese postman tour*.

As one can see that, if a digraph D is strongly connected and symmetric, it contains an Euler tour (see Lemma 2.1.1). Since an Euler tour contains each edge in D only once, it is therefore a Chinese postman tour as well. Thus, when D is strongly connected and symmetric, the Chinese postman problem can be reduced to that of finding Euler tour. However, if D is strongly connected but not symmetric, then a Chinese postman tour contains every edge in E at least once, but perhaps more than once. Given a postman tour T of D, let $\psi(v_i, u_j) \geq 1$ be the number of times edge (i, j) is contained in T. If, by replicating edge (i, j) $\psi(v_i, u_j)$ times, a symmetric digraph $D^{'}$ is obtained, and $D^{'}$ is called a *symmetric augmentation* of D. According to Theorem 2.1.1, an Euler tour exists in $D^{'}$. It is easy to prove that an Euler tour in $D^{'}$ is a Chinese postman tour in D if and only if the sum of the cost of replicated edges from the corresponding symmetric augmentation of D is minimal. Finding a Chinese postman tour in a digraph D is thus reduced to two steps:

1. augment D to derive a minimal symmetric digraph $D^{'}$;

2. find an Euler tour in $D^{'}$.

Construction of minimal symmetric augmentation can be accomplished by using a flow network. This has been discussed by Kuan (Kua62). Here, we describe the algorithm in overview.

Given a digraph $D = (V, E)$, the index of a vertex $v_i \in V$ is $\xi(v_i) = d^-_{v_i} - d^+_{v_i}$ where $d^+_{v_i}$ is the in-degree of v_i and $d^-_{v_i}$ the out-degree. Let $\{s, t\}$ be a set of vertices where s is the source and t the sink. Let E^+ and E^- be two sets of edges where $E^+ = \{(s, v_i) : \forall v_i \in V, b_{v_i} > 0\}$ and $E^- = \{(v_i, t) : \forall v_i \in V, b_{v_i} < 0\}$. A flow graph $D_f = (V_f, E_f)$ is constructed from D as follows: $V_f = V \cup \{s, t\}$ and $E_f = E \cup E^+ \cup E^-$.

Let each edge in E^+ and E^- has the cost of zero and capacity[1] $c(s, v_i) \equiv b_{v_i}$, $c(v_j, t) \equiv b_{v_j}$. The remaining edges in E_f have the same costs with their corresponding edges defined in D. For convenience, each edge in D is assigned a cost of 1. Each of the remaining edges has infinite capacity. A flow x on D_f is then a function $x : E_f \rightarrow \mathbb{Z}^+$ that satisfies the following conditions:

1. $\forall v_i \in V_f - \{s, t\}, \quad \sum_{(v_i, v_j \in E_f)} x(v_i, v_j) = \sum_{(v_j, v_i \in E_f)} x(v_j, v_i)$.
2. $\forall (v_i, v_j) \in E_f$, $x(v_i, v_j) \leq c(v_i, v_j)$ where $c(v_i, v_j)$ is the capacity of (v_i, v_j).

The cost of the flow x is given by

$$C(x) = \sum_{(v_i, v_j) \in E_f} C(v_i, v_j) x(v_i, v_j). \tag{2.6}$$

The problem is then converted to find a maximum-flow/minimum-cost flow x in D_f. Since x is a maximum flow and all edges in D has infinite capacity, all edges $(v_i, v_j) \in E(D)$, by replicating $\psi(v_i, v_j)$ times, saturate to s or t, namely, $x(s, v_i) = b_{v_i}, \forall v_i, b(v_i > 0)$ and $x(v_i, t) = b_{v_i}, \forall v_i, b(v_i < 0)$. The final augmented digraph D' is symmetric and, consequently, contains an Euler tour. Since the flow is also a minimum-cost flow, the number for replicating edges in D is minimal. Thus, the Euler tour in D' is a Chinese postman tour in D.

Lemma 2.1.4 *(ATLU91) An Euler tour P of a rural symmetric augmentation D' of D corresponds to a rural Chinese postman tour of D.*

Finding Chinese postman tour in a directed graph is of great value in the automated generation of test sequences in finite state machine based testing. This is discussed in Chapter 4.

2.2 Metaheuristic optimisation techniques

Optimisation has been attracting the interests of researchers for many years. In general, the problem is described as follows. Suppose $f(X)$ is a function with a set of parameters, $X = \{x_1, ..., x_m\}$, the problem is to find the set of X such that,

[1] It should be noted that edges in E^- have negative capacities since $b_{v_i} < 0$.

after applying X to the function, $f(X)$ is either maximised or minimised. Many algorithms have been proposed for solving the problem, among which Metaheuristic Optimisation Techniques (MOTs) such as Genetic Algorithms (GAs) (Gol89) and Simulated Annealing (SA) (MRR$^+$53) are used to find optimal solutions in the problems with a large search space.

Recently, MOTs have been introduced in software engineering for the generation of test data. Applications can be found in structural coverage testing (branch coverage testing) (JES98; MMS01), worst case and best case execution time estimating (WSJE97), and exception detecting (TCM98; TCMM00). In this section, some major MOTs are introduced. Testing with MOTs is reviewed in Chapter 3.

2.2.1 Genetic algorithms

Genetic Algorithms (GAs) (Gol89) work on the simulation of natural processes, utilising selection, crossover and mutation. Since Holland's seminal work (1975) (Hol75), they have been applied to a variety of learning and optimisation problems. Many versions of GAs have been proposed and they are all based on the simple GA.

I. Simple GA

A simple GA starts with a randomly generated population, each element (chromosome) being a sequence of variables/parameters for the optimisation problem. The set of chromosomes represents the search space: the set of potential solutions. The representation format of variable values is determined by the system under evaluation. It can be represented in binary, by real–numbers, by characters, etc. The search proceeds through a number of iterations. Each iteration is treated as a generation. At each iteration, the current set of candidates (the population) is used to produce a new population. The quality of each chromosome is determined by a fitness function that depends upon the problem considered. Those of high fitness have a greater probability of contributing to the new population.

Selection is applied to choose chromosomes from the current population and pairs them up as parents. *Crossover* and *mutation* are applied to produce new

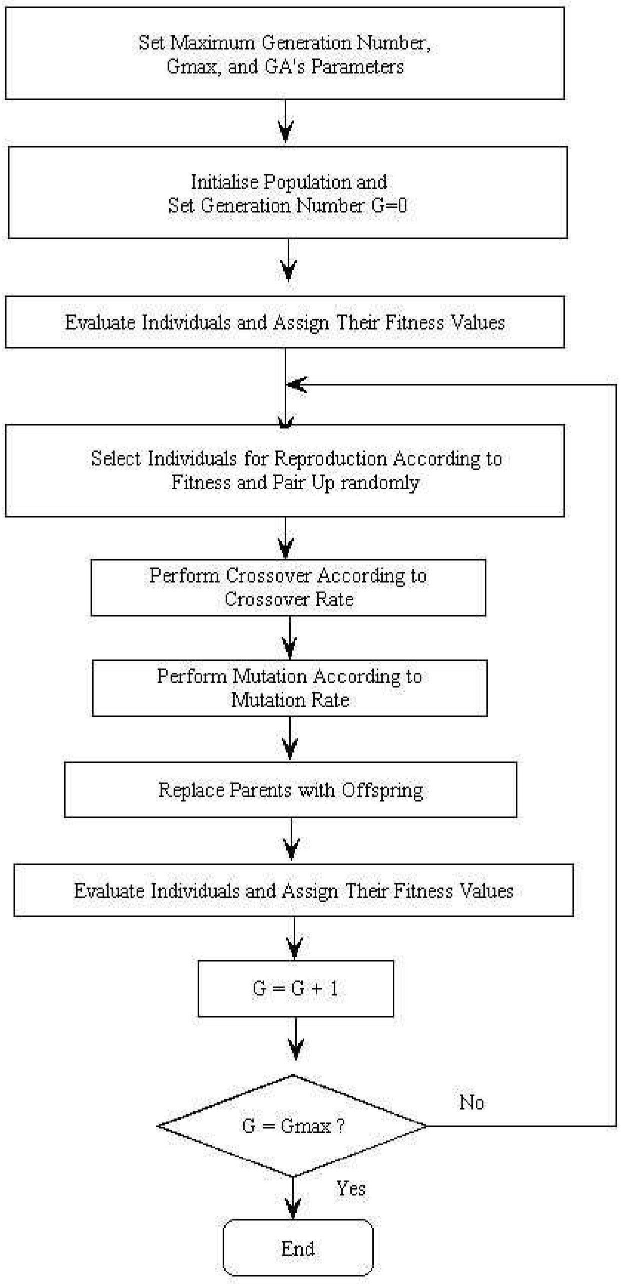

Set Maximum Generation Number, Gmax, and GA's Parameters
Initialise Population and Set Generation Number G=0
Evaluate Individuals and Assign Their Fitness Values
Select Individuals for Reproduction According to Fitness and Pair Up randomly
Perform Crossover According to Crossover Rate
Perform Mutation According to Mutation Rate
Replace Parents with Offspring
Evaluate Individuals and Assign Their Fitness Values
G = G + 1
G = Gmax ?
No
Yes
End

Figure 2.2: The flow chart of simple GA.

chromosomes. A new population is formed from new chromosomes produced on the basis of *crossover* and *mutation* and may also contain chromosomes from the previous population.

Figure 2.2 shows a flow chart for a simple GA. The following sections give a detailed explanation on *Selection*, *Crossover* and *Mutation*. All experiments in this work used roulette wheel selection and uniform crossover.

II. Encoding

In order to apply GAs, a potential solution to a problem should be represented as a set of parameters. These parameters are joined together to form a string of values (often referred to as a *chromosome*). Parameter values can be represented in various forms such as binary, real-numbers, characters, etc.

Obviously, the encoding strategy is central to the successful application of GAs. However, at present, there is no theory that enables a rigorous approach to the selection of the best encoding method for a particular problem. One principal that an encoding strategy needs to stick to is that the representation format should make the computation effective and convenient.

III. Reproduction

During the reproductive phase of a GA, individuals are selected from the population and recombined, producing children. Parents are selected randomly from the population using a scheme which favours the more fit individuals. Roulette Wheel Selection (RWS) and Tournament Selection (TS) are the two most popular selection regimes that are used for reproduction. RWS involves selecting individuals randomly but weighted as if they were chosen using a roulette wheel, where the amount of space allocated on the wheel to each individual is proportional to its fitness, while TS selects the fittest individual from a randomly chosen group of individuals.

Having selected two parents, their chromosomes are *recombined*, typically using the mechanisms of *crossover* and *mutation*. *Crossover* exchanges information between parent chromosomes by exchanging parameter values to form children. It takes two individuals, and cuts their chromosome strings at some randomly

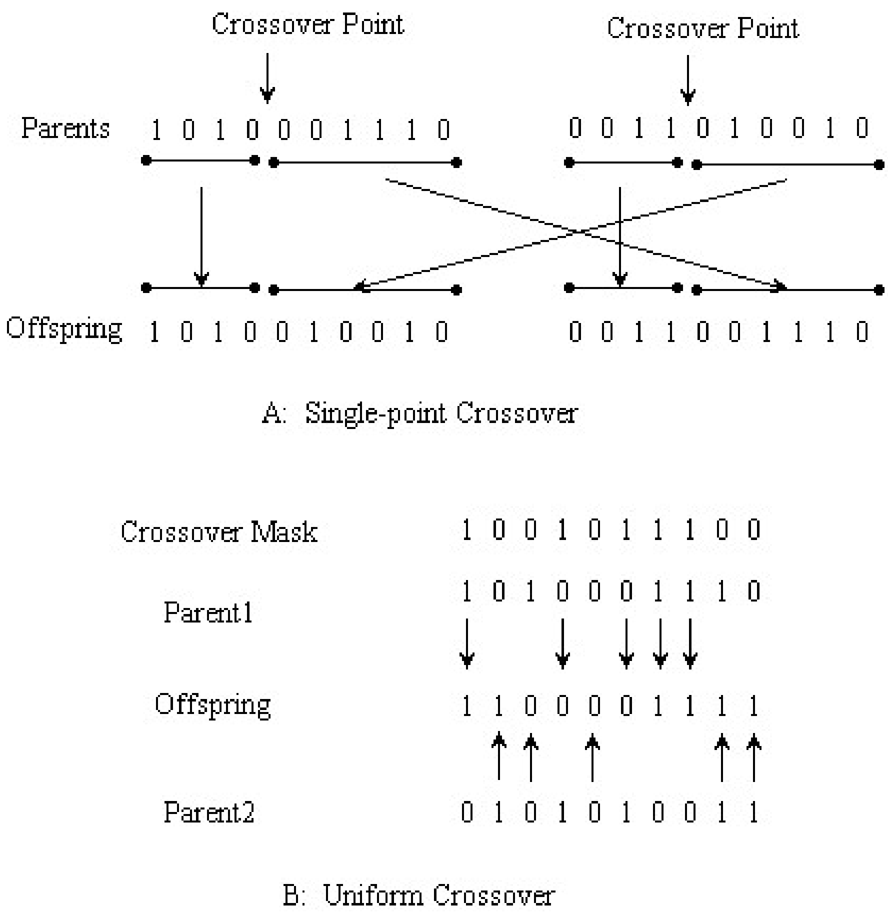


Figure 2.3: Crossover operation in simple GA.

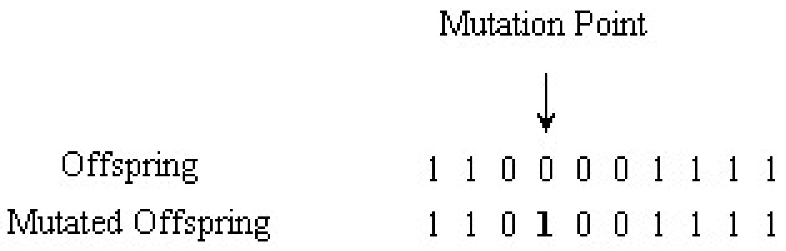


Figure 2.4: Mutation operation in simple GA.

chosen position, to produce two "head" segments, and two "tail" segments. The tail segments are then swapped over to produce two new full length chromosomes (see Figure 2.3 – A). Two offspring inherit some genes from each parent. This is known as *single point crossover*. In *uniform crossover*, each gene in the offspring is created by copying the corresponding gene from one or other parent, chosen according to a randomly generated *crossover mask*. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent (see Figure 2.3 – B). The process is repeated with the parents exchanged to produce the second offspring.

Crossover is not usually applied to all pairs of individuals selected for mating. A random choice is made, where the likelihood of crossover being applied is typically between 0.6 and 1.0 (Gol89). If crossover is not applied, offspring are produced simply by duplicating the parents. This gives each individual a chance of appearing in the next generation.

Mutation is applied to each child individually after crossover, randomly altering each gene with a small probability. Figure 2.4 shows the fourth gene of the chromosome being mutated. Mutation prevents the genetic pool from premature convergence, namely, getting stuck in local maxima/minima. However, too high a mutation rate prevents the genetic pool from convergence. A probability value between 0.01 and 0.1 for mutation is suggested (Gol89).

Elitism might be applied during the evolutionary computation. Elitism involves taking a number of the best individuals through to the next generation without subjecting them to selection, crossover and mutation. The number of individuals used for elitism is determined by $n(1-G)$ where n is the population size and G is the generation gap[1].

The use of elitism can significantly improve the performance of a GA for some problems. However, it should be noted that inappropriate settings for elitism might lead to premature convergence in the genetic pool.

IV. Sharing Scheme

A simple GA is likely to converge to a single peak, even in domains characterised by multiple peaks of equivalent fitness. Moreover, in dealing with multimodal functions with peaks of unequal value, the population of a GA is likely to crowd to the peak of the highest value. To identify multiple optima in the domain, some mechanisms should be used to force a GA to maintain a diverse population of members throughout its search. Sharing is such a mechanism that is proposed to overcome the above limitations. Sharing, proposed by Holland (Hol75) and expanded by Goldberg and Richardson (GR87), aims to reduce the fitness of individuals that have highly similar members within the population. This rewards individuals that uniquely exploit areas of the domain while discouraging redundant (highly similar) individuals in a domain. This causes population diversity pressure, which helps maintain population members at local optima.

The shared fitness of an individual i is given by $f_{(sh,i)} = \frac{f_{(i)}}{m_{(i)}}$, where $f_{(i)}$ is the raw fitness of the individual and $m_{(i)}$ is the peak count. The peak count is calculated by summing a sharing function over all members of the population $m_{(i)} = \sum_{j=i}^{N} sh(d_{(i,j)})$. The distance $d_{(i,j)}$ represents the distance between individual i and individual j in the population, determined by a similarity measurement. If the sharing function determines that the distance is within a fixed radius σ_{sh}, it returns a value determined by $sh(d_{(i,j)}) = 1 - (\frac{d_{(i,j)}}{\sigma_{sh}})^{\alpha_{sh}}$; otherwise it returns 0. α_{sh} is a constant that regulates the shape of the sharing function.

[1]Generation gap is the fraction of individuals replaced in evolving the next generation.

2.2.2 Simulated annealing

Simulated Annealing (SA) was first proposed by Metropolis *et al.* in 1953. It was originally proposed as a means of finding the equilibrium configuration of a collection of atoms at a given temperature. It was Kirkpatrick *et al.* (KGJV83) who suggested that a form of simulated annealing could be used for optimisation problems. The objective for the cooling of a material using a heat bath (a physical process known as *annealing*) is to cool the material slowly so that a near perfect lattice crystal structure (i.e. a state with minimum energy) is obtained. Through simulating such a process, the aim of simulated annealing is to iteratively improve a given solution by performing local changes. Changes that improve the solution are automatically accepted, whereas those changes that make the solution worse are accepted with a probability that depends on the temperature. Figure 2.5 shows the flow chart of simulated annealing algorithm.

Principally, the process of simulated annealing attempts to avoid local optima by allowing up-hill (or inferior) solutions in a controlled manner. The idea behind this is that "*it is better to accept a short-term penalty in the hope of finding significant rewards longer-term*" (KGJV83). In accepting an inferior solution, the search aims to escape from locally optimal solutions in order to find a better approximation to the global optimum. The control parameter (or temperature) is used to control the acceptance of inferior solutions. Acceptance of worse solutions depends upon the degree of inferiority and the current temperature. This degree is determined by a probability value calculated as $e^{-\frac{\Delta E}{T}}$ where ΔE is the absolute value of the difference in the objective function values between two solutions and T is a parameter analogous to the temperature.

The *cooling schedule* is crucial to the success of the search process. The selection of the values of iterations and temperatures is considered as using either a large number of iterations with a small number of temperatures or a small number of iterations with a larger number of temperatures. Generally, two cooling schemes are considered (Dow93). In the first case, a geometric reduction of the temperature by multiplication by a constant α is used. A value between 0.8 and 0.99 is usually suggested for α (KGJV83).

The number of iterations at each temperature in this scheme is usually increased geometrically or arithmetically. This allows intensive search at lower temperatures to ensure that local optima have been fully explored. An alternative to geometric increases in iterations is to use feedback from the annealing process. In this way, the time spent at high temperatures will be small and the time spent at low temperatures will be large. It is desirable to accept at least some solutions at each temperature. This aims to ensure that the neighbourhoods have been searched sufficiently. Once the temperature becomes lower, the number of accepted solutions may become so small that an infeasible number of iterations are required to accept the desired number of solutions, hence a maximum limit is normally also imposed. In the second cooling schedule, one iteration is performed at each temperature. However, the temperature is reduced extremely slowly in order that the exploration is sufficient. Selection of cooling schedule with reference to theoretical results of convergence to optimal solutions has been thoroughly studied by Dowsland (Dow93).

2.2.3 Others

Other MOTs, including *hill climbing* (RN95) and *tabu search*, have also been used for the problems of optimisation. Hill-climbing is essentially an iterative search where the value of the solution can only increase or stay the same at each step. In hill climbing, a randomly chosen point and its neighbours are considered for the search. Once a fitter neighbour is found, it becomes the 'current point' in the search space and the process is repeated; otherwise, if no fitter neighbour is found, then the search terminates and an optima has been found.

The seminal work on tabu search appears in (Glo89). It is a heuristic search technique based on the premise that problem solving, in order to qualify as intelligent, must incorporate adaptive memory and responsive exploration (Glo89). Thus, the algorithm of tabu search is based on that of the next k neighbours, while maintaining a tabu list (memory) that avoids repeating the search in the same area of the solution space. This is done by means of a tabu list of visited neighbours that are forbidden. Figure 2.6 illustrates the flow chart of tabu search algorithm.

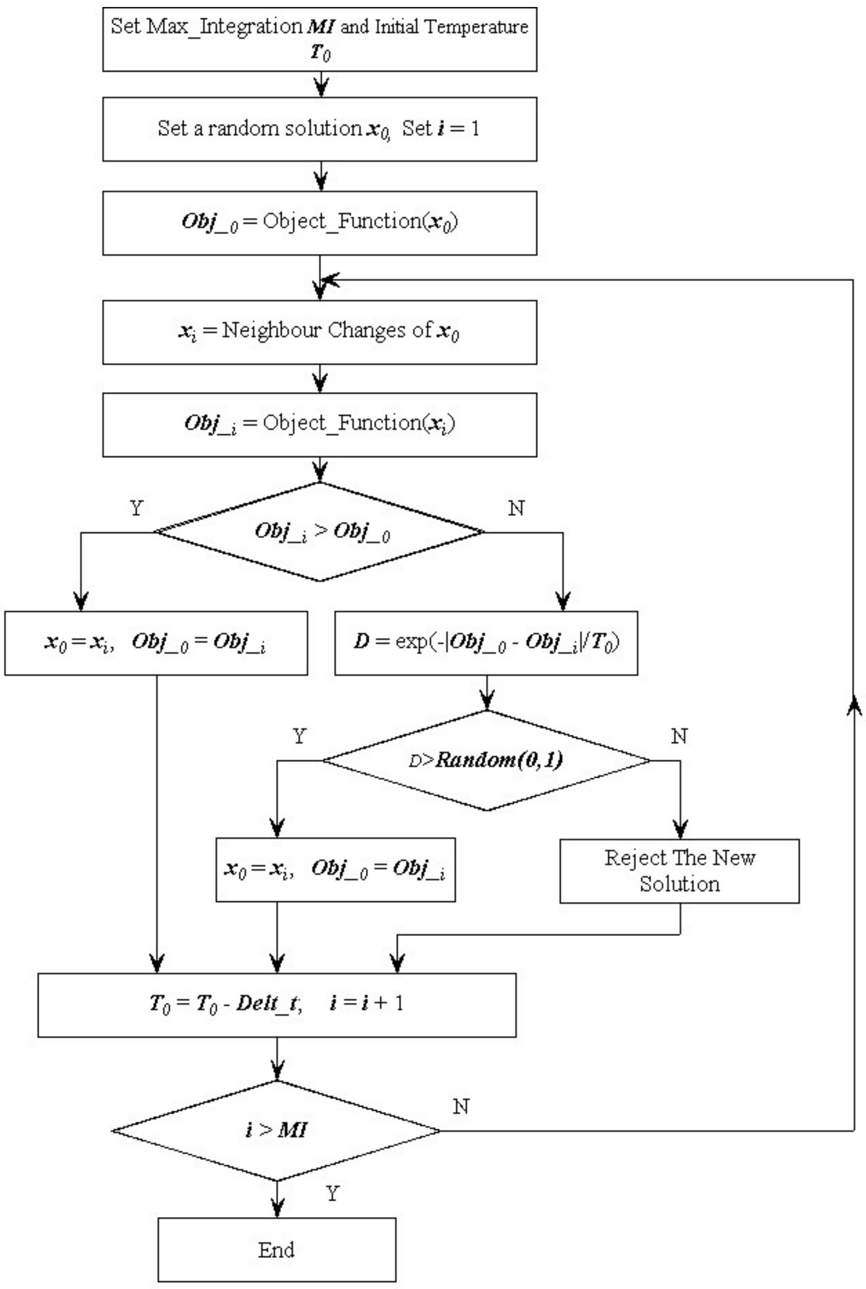

Set Max_Integration MI and Initial Temperature T0
Set a random solution x0, Set i = 1
Obj_0 = Object_Function(x0)
xi = Neighbour Changes of x0
Obj_i = Object_Function(xi)
Y
Obj_i > Obj_0
N
x0 = xi, Obj_0 = Obj_i
D = exp(-|Obj_0 - Obj_i|/T0)
Y
D>Random(0,1)
N
x0 = xi, Obj_0 = Obj_i
Reject The New Solution
T0 = T0 - Delt_t, i = i + 1
i > MI
N
Y
End

Figure 2.5: The flow chart of simulated annealing algorithm.

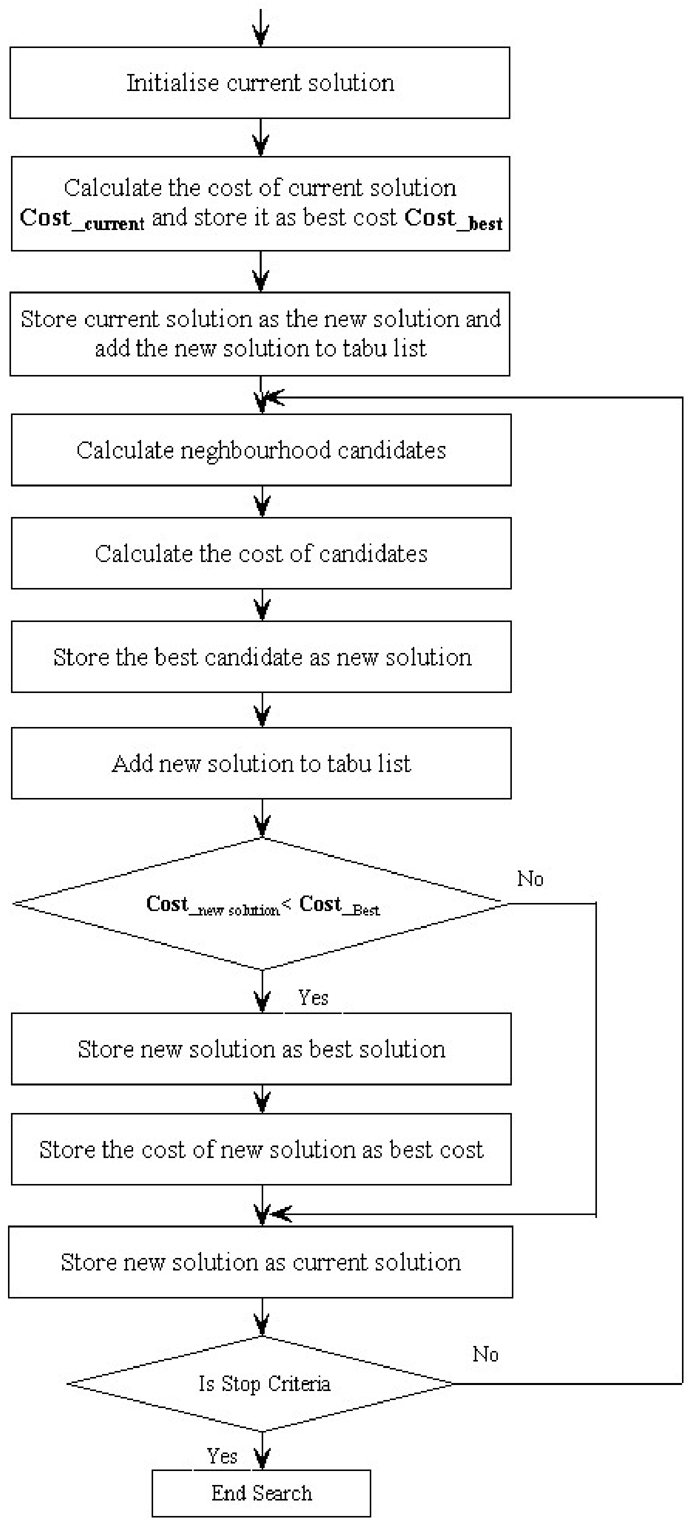

Initialise current solution
Calculate the cost of current solution Cost_current and store it as best cost Cost_best
Store current solution as the new solution and add the new solution to tabu list
Calculate neghbourhood candidates
Calculate the cost of candidates
Store the best candidate as new solution
Add new solution to tabu list
Cost_new solution< Cost_Best
No
Yes
Store new solution as best solution
Store the cost of new solution as best cost
Store new solution as current solution
Is Stop Criteria
No
Yes
End Search

Figure 2.6: The flow chart of tabu search algorithm.

Chapter 3

Test generation - a review

3.1 Introduction

Software needs to be adequately tested in order to ensure (at least to provide confidence) that the implementation under test conforms to its specification. Generation of test cases is thus required for the purpose of conformance testing. When testing a system, efficient test cases are always preferred. An efficient test case should cover all faults that the implementation may have and be relatively short.

Many approaches have been proposed for the generation of test cases. These approaches are either based upon looking at the program (code) being developed or rely on examining the specification that the implementation refers to. When an approach is selected for the generation of test cases, it is often supplemented by others since no approach guarantees to generate a complete test set. In this chapter, some of the main test generation techniques are reviewed. Finite state model based testing techniques are reviewed in chapter 4 separately.

3.2 Adequacy criteria

An adequacy criterion is a criterion that defines what constitutes an adequate test set. Adequacy criteria are essential to any testing methods as they provide measurements to justify a test set. Definitions of adequacy criteria might vary according to different test emphases. Criterion C_1 is said to subsume C_2 if and only if whenever a test set satisfies C_1, it satisfies C_2 as well.

Goodenough *et al.* (GG75) first studied test criteria and suggested that an adequacy criterion should be a predicate that defines "*what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested programs*". In order that the correctness of a program is adequately tested, Goodenough *et al.* proposed *reliability* and *validity* as properties to justify a test set TS. Reliability requires that a test criterion always produces consistent test results, while, validity requires that the test always produces meaningful results, namely, for every error in a program, there exists a test set that is capable of detecting this error.

Weyuker *et al.* (WO80) further studied test criteria and pointed out that these two properties are not independent. Since a criterion must either be valid or reliable, properties proposed by Goodenough *et al.* are mutually related.

When measuring a test set, an adequacy criterion can be defined in two ways. Firstly, an adequacy criterion can be used as a stopping rule to indicate whether more testing is needed. Secondly, instead of simply stating that a test set is good or bad, an adequacy criterion can be used to measure test quality by associating a degree of adequacy with each test set, namely, it not only directs the selection of test-data, but also decides the sufficiency of a given test set. Currently, two adequacy uses have been proposed for the evaluation of a test and they are defined as follows.

Definition 3.2.1 *(Test data adequacy criteria as stopping rules) (GG75). A test data adequacy criterion C is a function $C : P \times S \times T \rightarrow \{true, false\}$. $C(p, s, t) = true$ means that t is adequate for testing program p against specification s according to the criterion C, otherwise t is inadequate.*

Definition 3.2.2 *(Test data adequacy criteria as measurements) (GG75). A test data adequacy criterion is a function C, $C : P \times S \times T \rightarrow [0, 1]$. $C(p, s, t) = r$ means that the adequacy of testing the program p by the test set t with respect to the specification s is of degree r according to the criterion C. The greater the real number r, the more adequate the testing.*

It can be noted that these two uses are highly related. On one hand, the stopping rule can be described by the set $\{false, true\}$ where *true* suggests that the current test set is adequate enough, while, *false* implies more testing is required. The stopping rule can thus be viewed as a special case of measurement; on the other hand, given an adequacy measurement M and an adequacy degree d, it is always possible to define a stopping rule M_{stop} (determined by the test emphases) such that the adequacy degree d_t of a test set is no less than d, $d_t \geq d$. In terms of the value of d_t, measurement of a test set can be described by the set $\{false, true\}$. In many cases, these two uses are often mutually transformed from one to the other. By considering the two uses together, the adequacy criterion as a generator is then defined.

Definition 3.2.3 *(Test data adequacy criteria as generator) (BA82). A test data adequacy criterion C is a function $C : P \times S \rightarrow 2^T$ where D is the set of inputs of the program P and 2^T denotes the set of subsets of T. A test set $t \in C(p, s)$ means that t satisfies C with respect to p and s, and it is said that t is adequate for (p, s) according to C.*

3.3 Black-box and white-box testing

Two techniques are widely used in the generation of test cases, these being white-box testing and black-box testing. In black-box testing, the internal workings of the item being tested are not known by the tester, while, in white-box testing, explicit knowledge of the internal workings of the item being tested are used to select the test data.

It has been suggested that, due to some psychological facts, in the system development, the designer and the tester should be independent of each other (Bei90; Boe81). When testing a system, the tester receives very little knowledge of the internal implementation details of code and often views the system under test as a 'black box'. Tests are carried out by using the specification as a reference. Test cases are generated wholly from the specification, each of which aims to test some predefined functionalities. That is, given a set of inputs to the system under test, the only way to justify the correctness of outputs is to compare them to those defined in the specification. If differences are observed, faults in the system are then detected. The derivation of test cases in this testing is known as black-box testing. It is also referred to by some testers as functional testing.

One of the advantages of black-box testing is that the test is less likely to be biased since the designer and the tester are independent of each other. A tester can conduct the testing from the point of view of the user, not the designer. Test case design can therefore proceed once the specification is complete.

However, the disadvantages of black box testing can also be noted. If the tester derives a test case that has already been run by the designer, the test is redundant. In addition, the test cases are difficult to design. Since it is generally unrealistic to exhaustively test every possible input stream, some program units might go untested.

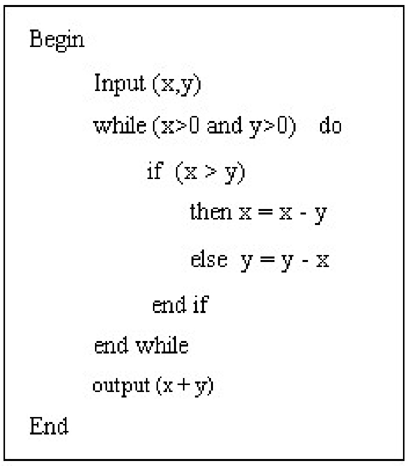
By contrast, it might be possible to look in detail at how test cases actually exercise particular elements of the implementation, namely, the code. For a given set of inputs, a program must execute some sequences of small execution steps in order to calculate the final outputs. Access to information about these steps and their effects allows more rigorous analysis of what the tests are going to achieve when the code is executed. This is referred to as white-box testing.

Some of the advantages on white-box testing can be noted. As the knowledge of internal coding structure is provided, it is easy to find out which type of input/data can help in testing the application effectively. Meanwhile, by looking at the internal structure of a program unit, white-box testing may help to optimise the code.

However, the use of white-box testing may also lead to some disadvantages. As knowledge of code and internal structure is a prerequisite, a skilled tester is needed to carry out this type of testing, which increases the cost of testing. At the same time, it is nearly impossible to look into every bit of code to find out hidden errors, which might results in the failure of fault detection in an application.

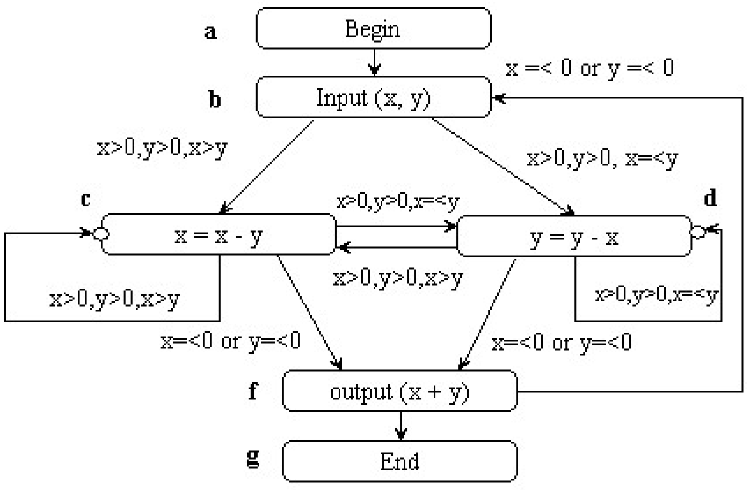
3.4 Control flow based testing

Control-flow based testing is based on the knowledge of the control structure of the program under test. It is a kind of white box based testing approach. The control structure of the program is usually represented by a control flow graph where a syntactic unit such as a predicate in a branch is represented by a node with edges that link this node to the nodes that are reachable through execution. Figure 3.1 illustrates an example of control flow graph.



```
Begin
        Input (x,y)
        while (x>0 and y>0)   do
               if  (x > y)
                     then x = x - y
                     else  y = y - x
               end if
        end while
        output (x + y)
End
```

A: Program source code



B: Control flow graph

Figure 3.1: An example of control flow graph.

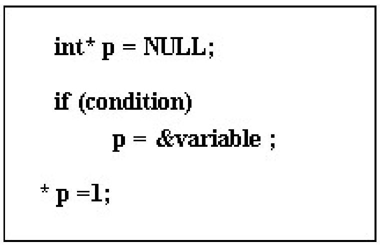
In the control flow graph, node n_j is called a post-conditioner of n_i if, when n_i is executed by an input $input_l$, n_j is reached; n_i is called the pre-conditioner of n_j. Two nodes n_i and n_j in the control flow graph can be merged as one node if and only if the following condition is satisfied: whenever n_i is executed, n_j is always a post-conditioner of n_i and whenever n_j is reached, n_i is always a pre-conditioner of n_j. After the control flow graph is accomplished, a *path* that starts at the first node (entrance of the program), traverses a sequence of edges and ends up at the terminal node (end of the program) can then be tested. Such a path is called a *computation path*, or an *execution path*. In order to achieve full testing coverage, all computation paths in the control flow graph need to be tested.

A variety of coverage criteria can be defined. Here, some major criteria are described.

Statement coverage

Statement coverage (Nta88; Bei90) reports whether all executable statements in the program have been encountered. Statement coverage selects a set of test cases TS such that, by executing a program P with each test case $ts_i \in TS$, all statements of P (nodes in the control graph) have been executed at least once.

The advantage of this measure is that it can be directly applied to object code and does not require processing source code. However, statement coverage is subject to a disadvantage where the measurement is insensitive to some control structures. For example, in the C/C++ code shown in Figure 3.2, without a test case that causes condition to evaluate false, statement coverage rates this code fully covered. In fact, if condition ever evaluates false, this code fails.



```
int* p = NULL;

if (condition)
        p = &variable ;

* p =1;
```

Figure 3.2: An example of the statement coverage.

Branch coverage

Branch coverage (Nta88) measures the coverage of all blocks and statements that affect the control flow. In a statement, boolean expressions are evaluated for both *true* and *false* conditions. Branch coverage criterion selects a set of test cases TS such that, by executing a program P with each test case $ts_i \in TS$, all edges in the control graph have been traversed at least once.

It can be noted that branch coverage subsumes statement coverage because if all edges in a control flow graph are covered, all nodes (statements) are correspondingly covered. Therefore, if a test set satisfies the branch coverage, it also satisfies the statement coverage.

However, the use of branch coverage may lead to some conditions within boolean expressions, or relevant combinations of conditions being ignored, which leads to an incomplete test.

Condition coverage

Condition coverage (Bei90) measures the sub-expressions independently of each other, which allows for a better analysis of the control flow. Condition coverage criterion selects a set of test cases TS such that, by executing a program P with each test case $ts_i \in TS$, all edges in the control graph have been traversed at least once and all possible values of the constituents of compound conditions have been exercised at least once.

There are two strong versions of condition coverage, these being *multiple condition coverage* (WHH80) and *modified condition/decision coverage* (MC/DC) (CM94). Multiple condition coverage reports whether every possible combination of boolean sub-expressions has been examined, while, MC/DC requires that every condition that can affect the result of its encompassing decision needs to be verified at least once.

With condition coverage, the sub-expressions are combined by logic operator "AND" and "OR" respectively. Test cases required for full multiple condition coverage of a condition are given by the logical operator truth table for the condition. This is how multiple condition coverage works for the generation of test cases. An advantage of multiple condition coverage is that it requires very thorough testing, which, in theory, makes it the most desirable structural coverage measure. However, it may be noted that it is difficult to determine the minimum set of test cases required to achieve the test goal, especially for those boolean expressions with high complexity. It has been suggested that, for a decision with

n conditions, multiple condition coverage requires 2^n tests (WHH80). This makes the multiple condition an impractical coverage criterion. In addition, the number of test cases required could vary substantially among conditions that have similar complexity.

MC/DC was first created at Boeing for the use of testing aviation software. MC/DC requires that each condition be shown to independently affect the outcome of the decision. The independence requirement ensures that the effect of each condition is tested relative to the other conditions. MC/DC is a strong coverage criterion since a thorough execution of the code is required. It subsumes the branch and statement coverage criteria. However, achieving MC/DC requires more thoughtful selection of the test cases, and, in general, for a decision with n conditions, a minimum of $n+1$ test cases is required (CM94). MC/DC is actually a mandatory criterion for software developed for critical applications in the civil aerospace industry (CM94).

Path coverage

Path coverage measures the percentage of all possible paths through the program being tested. It selects a set of test cases TS such that, by executing a program P with each test case $ts_i \in TS$, all paths starting from the initial node of the control flow graph of P have been traversed at least once.

It can be noted that, although the path coverage criterion cannot guarantee program correctness, it is a strong criterion as very thorough testing is required.

The main drawback of the path coverage criterion is that the number of paths is exponential in the number of branches. Moreover, the number of pathes can be infinite if a path is a loop. It is also nearly unrealistic to test all possible paths if the program under test has a large size.

3.5 Data flow based testing

Data-flow based testing (RW85) mainly focuses on the investigation of how values are associated with variables in a program P and how these associations affect the execution of the program. In a program, a variable that appears in a statement can be classified as either a definition occurrence or a use occurrence. The definition occurrence of a variable determines that a value is required to be assigned to the variable, while, the occurrence of use indicates that the value of the variable is referred.

A use occurrence of a variable can be classified as two uses - the *computational use* and the *predicate use*. If the value of a variable is used to produce *true* or *false* for a predicate, the occurrence of the variable is called *predicate use*; otherwise, if it is used to compute a value for other variables or as an output value, it is called a *computational use*. For example, statement $y = x_1 + x_2$ requires the values of x_1 and x_2 to produce the outcome (definition) of y. In contrast, statement "if $x_1 < x_2$ then goto L endif" contains a predicate that uses the values of x_1 and x_2 as references.

Three families of adequacy criteria have been proposed for data-flow based testing. Before introducing these test criteria, some definitions are introduced.

Definition 3.5.1 *A variable x is defined if it is declared or assigned to or contained in an input statement.*

Definition 3.5.2 *A variable x is computation-used if it 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.*

Definition 3.5.3 *A variable x is predicate-used if it forms part of a predicate in a conditional-branch statement.*

Definition 3.5.4 *A definition free path with respect to variable x is a path where for all nodes in the path there is no definition occurrence of x.*

Definition 3.5.5 *A path is cycle-free if all visited nodes are distinct.*

Lemma 3.5.1 *(RW85) A definition occurrence of a variable x at a node u reaches a computational use occurrence of the variable at node v if and only if there is a path p from u to v such that $p = (u, w_1, w_2, ..., w_n, v)$, and $(w_1, w_2, ..., w_n)$ is definition free with respect to x and the occurrence of x at v is a computational use.*

Three adequacy criteria

C_1: The Rapps-Weyuker-Frankl criteria

Based upon data-flow information, Rapps *et al.* (RW85) proposed a class of testing adequacy criteria that mainly focus on the analysis of the simplest type of data-flow paths that start with a definition of a variable and terminate with a use of the same variable.

Frankl *et al.* (FW88) reexamined the criteria and found that the original definitions of the criteria did not satisfy the applicability property[1]. They then modified the definitions and proposed the so-called *all-definitions criterion*.

Definition 3.5.6 (All-definitions criterion) *A set P of execution paths satisfies the all-definitions criterion if and only if for all definition occurrences of a variable x such that there is a use of x which is feasibly reachable from the definition, there is at least one path p in P such that p includes a subpath through which the definition of x reaches some use occurrence of x.*

The all-definition criterion requires that an adequate test set should cover all definition occurrences in the sense that, for each definition occurrence, the testing paths should contain a path through which the definition reaches a use of the definition.

Herman (Her76) studied the data flow information and proposed the *all-uses criterion* (also called *reach-coverage* criterion).

Definition 3.5.7 (All uses criterion) *A set P of execution paths satisfies the all-uses criterion if and only if for all definition occurrences of a variable x and all uses occurrences of x that the definition feasibly reaches, there is at least one path p in P such that p includes a subpath through which that definition reaches the use.*

Since one definition occurrence of a variable may reach more than one use occurrence, the all-uses criterion requires that all of the uses should be exercised by testing. This makes the all-uses criterion stronger than the all-definition criterion.

[1]An adequacy criterion C satisfies the applicability property if and only if for every program P there exists some test set which is C-adequate for P (FW88).

A weakness of the above criteria was noticed by Frankl *et al.* (FW88) and Clarke *et al.* (CPR89). Given a definition occurrence of a variable x and a use of x that is reachable from this definition, there may exist more than one path through which the definition reaches the use. However, the criteria proposed above require only one of such paths to be exercised by testing. If all paths are to be exercised, the testing might be infinite since there may exist infinite such paths in the flow graph. To overcome this problem, Frankl *et al.* and Clarke *et al.* restricted the paths to be cycle-free or only the end node of the path to be the same as the start node, and then proposed the *all-definition-use-paths criterion.*

Definition 3.5.8 (All definition-use-paths criterion) *A set P of execution paths satisfies the all definition-use-paths criterion if and only if for all definitions of a variable x and all paths q through which that definition reaches a use of x, there is at least one path p in P such that q is subpath of p, and q is cycle-free or contains only simple cycles.*

C_2: The Ntafos required k-tuples criteria

Ntafos (Nta84) studied the interactions among variables from data flow and proposed a class of adequacy criteria called *required k-tuples* where $k > 1$ is a natural number. In the data flow graph, chains of alternating definitions and uses are called definition-reference interactions (abbreviated as $k - dr$ interactions). Ntafos' criteria require a path set that covers the $k - dr$ interactions.

Definition 3.5.9 *(Nta88) For $k > 1$, a $k - dr$ interaction is a sequence $K = [d_1(x_1), u_1(x_1), d_2(x_2), u_2(x_2), ..., d_k(x_k), u_k(x_k)]$ where*

1. *$d_i(x_i)$, $1 \leq i \leq k$, is a definition occurrence of the variable x_i;*
2. *$u_i(x_i)$, $1 \leq i \leq k$, is a use occurrence of the variable x_i;*
3. *the use $u_i(x_i)$ and the definition $d_{i+1}(x_i)$ are associated with the same node n_{i+1};*
4. *for all i, $1 \leq i \leq k$. the i^{th} definition $d_i(x_i)$ reaches the i^{th} use $u_i(x_i)$.*

Definition 3.5.10 *An interaction path for a $k-dr$ interaction is a path $p = ((n_1)*p_1*(n_2)*...*(n_{k-1})*p_{k-1}*(n_k))$ such that for all $i = 1, 2, ..., k-1$, $d_i(x_i)$ reaches $u_i(x_i)$ through p_i. n_i is the node that associates the use $u_i(x_i)$ with the definition $d_{i+1}(x_i)$; p_i is a definition-clear path from n_i to n_{i+1}.*

Clark *et al.* (CPR89) noted that variables and nodes used in definition 3.5.9 need not be distinct. They then proposed a more useful definition where distinction of variables and nodes is required.

Definition 3.5.11 (Required k-tuples criteria) *A set P of execution paths satisfies the required k-tuples criterion, $k > 1$, if and only if for all $j-dr$ interactions L, $1 < j \leq k$, there is at least one path p in P such that p includes a subpath which is an interaction path for L.*

C_3: The Laski-Korel criteria

Laski and Korel (LK83) studied the data flow and observed that a given node may contain uses of several different variables, each of which may be reached by some definitions occurring at different nodes. Such definitions constitute the computational context of that node. Each node in the flow graph can then be tested with contexts explored by selecting paths along which the various combinations of definitions that reach the current node. Based upon this observation, Laski and Korel proposed *context coverage criterion.*

Definition 3.5.12 (Ordered-context) *Let n be a node in the flow graph. Suppose that there are uses of the variables $x_1, x_2, ..., x_m$ at the node n. Let $[n_1, n_2, ..., n_m]$ be a sequence of nodes such that for all $i = 1, 2, ..., m$, there is a definition of x_i on node n_i and the definition of x_i reaches the node n with respect to x_i. A path $p = p_1*(n_1)*p_2*(n_2)*...*p_m*(n_m)*p_{m+1}*(n)$ is called an ordered context path for the node n with respect to the sequence $[n_1, n_2, ..., n_m]$ if and only if for all $i = 2, 3, ..., m$, the subpath $p_i*(n_i)*p_{i+1}*(n_{i+1})*...*p_{m+1}$ is definition free with respect to x_{i-1}. In this case, the sequence $[n_1, n_2, ..., n_m]$ of nodes is an ordered context for n.*

Definition 3.5.13 (Ordered-context coverage criterion) *A set P of execution paths satisfies the ordered-context coverage criterion if and only if for all nodes n and all ordered contexts c for n, there is at least one path p in P such that p contains a subpath which is an ordered context path for n with respect to c.*

Definition 3.5.14 (Context coverage criterion) *A set P of execution paths satisfies the context coverage criterion if and only if for all nodes n and for all contexts for n, there is at least one path p in P such that p contains a subpath that is a definition context path for n with respect to the context.*

3.6 Partition analysis

Partition analysis method (OB89) aims to generate a test set that checks programs on certain error-prone points.

Generally speaking, program errors may fall in two types: *computation errors* and *domain errors*. A computation error is reflected by an incorrect function in the program. Such an error may be caused, for example, by the execution of an inappropriate assignment statement that affects the computation outcome of the function within a path in the program. Domain errors are faults caused by the incorrect selection of boundaries for a sub-domain. A domain error may occur, for instance, if a branch predicate is incorrectly expressed, or an assignment statement that affects a branch predicate is wrong, which will affect the conditions under which the path is selected.

The partition analysis method works on partitioning the input space into subdomains and then select a small number of test (usually one) from each of these subdomains, aiming to find any computation errors in the subdomains. Each subdomain is defined so that the inputs it contains are treated similarly by the program, in some sense. It is assumed that this similarity makes it likely that if the program fails on one input in a subdomain, it also fails on a significant portion of the others. When testing with partition analysis, in order that the test effort is reduced, only a few representatives are selected from each subdomain for testing.

Three strategies have been proposed for the partitions of the input space. They are discussed in the following.

3.6.1 Specification based input space partitioning

Specification based input space partitioning considers the use of a subset of data as a subdomain if the specification requires the same function on the data. An example of specification based input space partitioning is illustrated in (HH91) where the function of a module called DISCOUNT INVOICE is described as follows: Two products, X and Y, are under sales with the single price of \$5 for X and \$10 for Y. A discount of 5% will be approved if the total purchasing is greater than \$200. If the total purchasing is greater than \$1,000, a discount of 20% is given. Produce X is encouraged for sales where if more than 30 Xs are purchased, a further discount of 10% is given. In the final calculation, non-integer costs are rounded down to give an integer value.

DISCOUNT INVOICE module has properties of $X \leq 30$ and $5*X+10*Y \leq 200$, and the output is calculated as $5*X+10*Y$. That is $\forall(X,Y) \in \{(X,Y)|x \leq 30, 5*X+10*Y\}$, $output \equiv 5*X+10*Y$. Subset $\{(X,Y)|x \leq 30, 5*X+10*Y\}$ should then be treated as one subdomain. In figure 3.3, partition of the input space of DISCOUNT INVOICE module is illustrated. Six subdomains are defined.

The above specification is written informally. In most cases, it is hard to derive partitions from an informal specification. However, if a specification is written with formal specification languages and the specification is in certain normal forms, it is always possible to derive partitions.

Hierons (Hie93) proposed a set of transformation rules where specifications written in pre/postconditions are transformed into the following normal form.

$$\begin{gathered}
P_1(x_1,x_2,...,x_n) \wedge Q_1(x_1,x_2,...,x_n,y_1,y_2,...,y_m) \vee \\
P_2(x_1,x_2,...,x_n) \wedge Q_2(x_1,x_2,...,x_n,y_1,y_2,...,y_m) \vee \\
\dots \\
P_K(x_1,x_2,...,x_n) \wedge Q_K(x_1,x_2,...,x_n,y_1,y_2,...,y_m)
\end{gathered}$$

where $P_i(x_1,x_2,...,x_n)$, $i=1,2,...,K$, are preconditions that give the condition on the valid input data and the state before the operation, and $Q_i(x_1,x_2,...,x_n, y_1,y_2,...,y_m)$, $i=1,2,...,K$, are post-conditions that specify the relationship

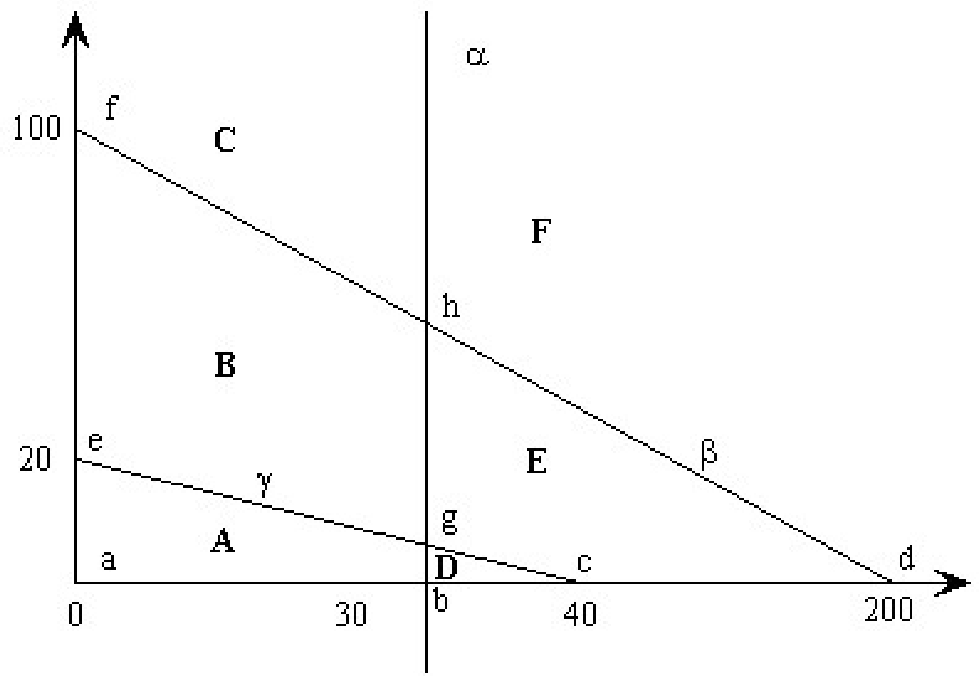


Figure 3.3: Partition of the input space of DISCOUNT INVOICE module. α, β, γ: borders of the subdomains; a,b,...,h: vertices of the subdomains; A,B,...,F: subdomains.

between the input data, output data, and the state before and after the operation. Variables x_i are input variables and y_i are output variables.

3.6.2 Program based input space partitioning

The input space can be partitioned according to the program structure. In this type of partitioning, two input data in a subdomain usually execute the same path in the program. In the program, a path often requires some inputs to trigger the condition for the execution. The condition is called *path condition* (How76). A path condition can be derived by symbolic execution (How76).

For example, in the DISCOUNT INVOICE module, there are six paths in the program, each of which requires an input to trigger the condition for the execution of this path. The partitions of input space are therefore determined by path conditions. Each path in the program is defined as a subdomain.

Testing with program based partitioning shows some similarities to path coverage based testing. If only one test case is required and the position of the subdomain is not considered, then testing with program based partitioning is equivalent to path coverage based testing. However, usually, partitioning testing requires test cases selected not only within the subdomains, but also on the boundaries, at vertices *etc* as these points are thought to be error-prone. A test case in the subdomain is called an *on* test point; otherwise, an *off* test point.

3.6.3 Boundary analysis

White *et al.* (WC80) proposed a test method called $N \times 1$ domain-testing strategy where N test cases need to be selected on the borders in an $N-$dimensional space and one case is just off the border. Clarke *et al.* (CHR82) extended the method to $N \times N$ criterion where, instead of one test case, N test cases are required to be off the border. Moreover, the N test cases should be linearly independent.

Definition 3.6.1 ($N \times 1$ domain adequacy) *(WC80) Let $\{D_1, D_2, ..., D_n\}$ be the set of subdomains of software S that has N input variables. A set T of test cases is said to be $N \times 1$ domain-test adequate if, for each subdomain D_i, $i = 1, 2, ..., n$, and each border B of D_i, there are at least N test cases on the border B and at least one test case that is just off B. If the border is in the domain D_i, the test case off the border should be an off test point; otherwise, the test case should be an on test point.*

Definition 3.6.2 ($N \times N$ domain adequacy) *(CHR82) Let $\{D_1, D_2, ..., D_n\}$ be the set of subdomains of software S that has N input variables. A set T of test cases is said to be $N \times N$ domain-test adequate if, for each subdomain D_i, $i = 1, 2, ..., n$, and each border B of D_i, there are at least N test cases on the border B and at least N linearly independent test case that is just off B. If the border is in the domain D_i, the N test case off the border should be an off test point; otherwise, the test case should be an on test point.*

It can be seen that boundary analysis focuses on testing the borders of a subdomain. The $N \times 1$ domain-testing strategy aims to check whether there exist parallel shift errors in a border, while, $N \times N$ domain-testing strategy checks not only parallel shift but also rotation of linear borders.

A special case can be noted in boundary analysis, this being vertex testing. Vertices are intersection points of borders. Clarke *et al.* (CHR82) suggested that vertices need to be used as test cases to improve the efficiency of boundary analysis. She also proposed the adequacy criterion for vertex testing defined as follows.

Definition 3.6.3 ($V \times V$ domain adequacy) *(CHR82) Let $\{D_1, D_2, ..., D_n\}$ be the set of subdomains of software S. A set T of test cases is said to be $V \times V$ domain-test adequate if, for each subdomain D_i, $i = 1, 2, ..., n$, T contains the vertices of D_i and for each vertex v of D_i, there is a test case just off v. If a vertex v of D_i is in the subdomain D_i, then the test case just off v should be an off test point; otherwise, it should be an on point.*

3.7 Mutation testing

Mutation testing (DLS78) is a fault-based testing technique that mainly focuses on measuring the quality of a test set according to the ability to detect specific faults. In mutation testing, a number of simple faults, such as simply altered operators, constant values and variables, are artifactually injected into the program under test one at a time. These modified programs are called *mutants*. A set of test cases is then designed, aiming to distinguish each mutant from the original program by the program outputs. If a mutant can be distinguished from the original program by at least one test case in the test set, the mutant is *killed*; otherwise the mutant is *alive*.

An example is illustrated in Figure 3.4 where the original code is $z = x+y$ and two mutants are generated by altering the arithmetic operator "+" to "-" and "*" respectively. Test case $(x = 0, y = 0)$ kills none of the mutants as the output z will be the same for the original and mutant programs. Test case $(x = 2, y = 2)$ kills mutant 1 but fails to kill mutant 2 as, when applying the input, mutant 1 produces an output that is different from that produced by the original program, while, mutant 2 produces the same output as the original program does. Mutant 1 is then killed while mutant 2 is still alive.

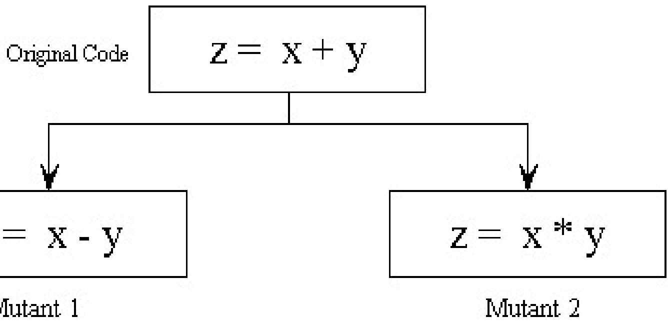


Figure 3.4: An example of mutation testing.

Mutation testing is based on two assumptions, namely, the competent programmer hypothesis and the coupling effect (DLS78). The competent programmer hypothesis assumes that programmers create programs that are close to being correct, namely, programmers only make small errors in the programs. This is the reason why a mutant is generated by deviating the original program slightly rather then considerably. The coupling effect assumes that a set of test cases that is capable of finding all simple faults in a program is also able to detect more complex faults. This assumption assures that the ability of the test set is not limited to recognising only simple faults.

Sometimes, however, a mutant cannot be killed due to the equivalence of the mutant and the original program. This leads to the studies of the adequacy of a test set. The adequacy of a test set is assessed by equation 3.1 where M is the total number of mutants, D is the number of mutants that has been killed and E is the number of equivalent mutants.

$$Adequacy = \frac{D}{M - E}. \tag{3.1}$$

The problem of deciding whether a mutant is equivalent to the original program (determining E in equation 3.1) is generally undecidable. The equivalence of a mutant is mostly determined manually. More recently, metaheuristic optimisation techniques have been suggested for the elimination of equivalent mutants. The related work can be found in ref. (AHH04).

One advantage of mutation testing is that it allows a great degree of automation. Once a set of mutation operators is carefully defined, the generation of mutants can be automated. Since the execution of the original program and the set of generated mutants can be automated, the comparison of the results can be automated.

However, mutation testing may lead to a high computational cost. It has been estimated that the number of mutants that can be generated is of the order of N^2 for an N-line program (How82). It might also require expensive human effort to identify equivalent mutants. Meanwhile, mutation testing relies on two hypotheses and this might require substantial empirical studies to validate the correctness of the hypotheses. In addition, mutation testing is still a technique that largely works on unit test. Instead of being independently used in testing applications, mutation testing remains a supplement that provides a means to evaluate the effectiveness of other testing techniques.

Based upon the same idea of mutation analysis, several variants are proposed. Howden (How82) proposed *weak mutation testing* to improve the test efficiency. In the original mutation testing (referred to as *strong mutation testing*), a change to a program is made before the execution, and the change is not reversed before the executions are complete. The outcomes of the original program and the mutant are compared only when the executions are finished, and the comparison is made upon the outputs of the two; in weak mutation testing, a component in a program is mutated. A test set that passes through this component is generated. If, when applying the test set, the mutated component produces a different value for a variable than the original one, then this mutant is killed.

The main advantage of weak mutation testing is that it is easier to generate test cases to kill mutants, which can improve the test efficiency. However, weak mutation testing only examines the mutated component. This may lead to a lower level of confidence. Compared to strong mutation testing, weak mutation testing is inferior.

Woodward *et al.* (WH88) proposed *firm mutation testing*. Firm mutation testing makes a compromise between strong mutation testing and weak mutation testing. In firm mutation testing, a tester is allowed to make a decision on when to compare the values between a component and its mutant. It can be set to be immediately after each single execution of the component, or at some execution points in the program. A tester is also allowed to decide how to compare the outcomes, for example, the output values, or the execution traces, and so on.

Zeil (Zei83) proposed *perturbation testing* where an “error” space is considered for the analysis of test effectiveness. Perturbation testing is quite similar to mutation testing. It is mainly concerned with faults in arithmetic expressions with program statements. Perturbation testing can be viewed as a special version of mutation testing.

3.8 Statistical testing

So far, all testing methods discussed are aimed at fault detection, with the goal of correctness. This, however, is not the only motivation for testing. Given an extensively tested implementation, if no failure is revealed, it suggests that there is a higher level of confidence in the system than before the testing is carried out; otherwise, faults are detected in the system under test. Faults might be categorised into several types, each of which has some features. If the probability of a type of fault can be estimated, it would be of great value in improving the process of testing. Statistical testing is thus proposed to achieve such a goal.

Statistical testing generates test cases using random number generating process. When devising a statistical testing strategy, an input space is defined for test data sampling. The input space defines the set of all possible inputs where an input defines the set of all variables needed for the calculation of an output. Test cases must be statistically independent. That is, the next test case chosen must not be influenced by the history of previously executed tests (Ehr89).

One widely used statistical approach is random testing. It is a form of functional testing. In random testing, test cases are selected randomly from the entire input domain of the program. When generating a test case, a weight value may be used to control the distribution of the selected data, for example, uniform distribution.

The effectiveness of random testing has been studied by several investigators. However, conclusions drawn from the studies varied significantly. Duran *et al.* (DN84) compared random testing with domain partitioning testing and found that, under the condition where the failure rates in the subdomains were either close to 0 or close to 1 and the subdomans are of equal size, the adequacies of random testing and partitioning testing are close. Miller *et al.* (MMN$^+$92) further studied random testing and described circumstances where random testing can increase confidence. Hamlet *et al.* (HT90) compared the random testing with partitioning testing by considering boundary cases where there exist hidden subdomains, subdomains that random testing is less likely to hit, and found that partitioning testing is far better than random testing. This study suggests that the effectiveness of partitioning testing and random testing is heavily affected by the expected failure rates of boundary cases.

The principal advantage of random testing is that it is comparatively simple and involves little effort in the generation of test cases. When automating the process of random testing, the use of a test oracle is usually not a requirement. In addition, random testing provides supplements for other testing techniques. As introduced in the previous chapters, testing is a complicated process and no testing technique guarantees to generate a complete test. Random testing can be used to further measure the adequacy of other techniques.

One of the big problems of random testing is when to stop testing. The other problem of random testing is to know when a test fails (DN84). It is hard to determine if the test cases generated with random testing has completely tested the system under test. To overcome these drawbacks, it is advisable to use some other techniques to check the adequacy of these test sets.

3.9 Search-based testing

Search based optimisation techniques such as Genetic Algorithms (GAs) and Simulated Annealing (SA) (see chapter 2) have recently been applied in the generation of test cases. In order that search based techniques can be applied, an objective f_{obj} function is required to evaluate the quality of a test case. A set of test cases is iteratively updated to explore the test cases that maximise/minimise f_{obj}.

The reasons that search based techniques are used in the generation of test cases are: (1) the problem of generating test cases is equivalent to a search problem where good solutions need to be explored in the input space of the system being developed. Usually, the input space has a large size. This could make the search a costly and inefficient process when traditional algorithms are applied. This problem, however, can be alleviated by heuristic search; (2) some problems in testing are NP-hard and search based techniques have proved efficient in providing good solutions for NP-problems.

Jones *et al.* (JES98) applied GAs for structural coverage testing (branch coverage testing). In the work, the control flow of the program is created. A fitness function is defined to guide the search of test data. This fitness function is based on the predicate associated with each branch, and a value for fitness derived from either a Hamming distance or a simple numerical reciprocal function. The simulation results suggested the test data generated are of high quality. Michael *et al.* (MMS01) studied the automated generation of dynamic test data by using GAs. A tool called GADGET (the Genetic Algorithm Data GEneration Tool) was devised for exploring test data. This tool allows the test generator to slightly modify the input parameters of the program, attempting to lead them to the values that satisfy the test requirement.

Tracey *et al.* (TCM98; TCMM00) investigated exception detection using GAs and SA respectively. Exception is a sub-class of failures. It might be caused by incorrect inputs, hardware faults or logical errors in the software code. In the work, a fitness function (see table 3.1) is defined to justify the test data that might raise exceptions. This fitness function provides a measure of how close a test case is to execute a desired raise statement. The process of the generation of such test data is further optimised by using GAs. The experimental results show the effectiveness of the proposed method.

Wegner *et al.* (WSJE97) investigated worst case and best case execution time estimation by using GAs. In the work, based upon the execution time measured in processor cycles, a fitness function is defined. A number of programs are used for experiments. By checking the experimental results, Wegner *et al.* claimed that GAs are able to check large programs. At the same time, they show considerable promise in establishing the validity of the temporal behaviour of real-time software.

However, the use of search based techniques for the generation of test cases has some limitations as well. Usually, in search based testing, inputs of the program

Element	Value
Boolean	if TRUE then 0 else K

$a = b$	if $abs(a - b) = 0$ then 0
	else $abs(a - b) + K$

$a \neq b$	if $abs(a - b) \neq 0$ then 0
	else K

$a < b$	if $a - b < 0$ then 0
	else $(a - b) + K$

$a \leq b$	if $a - b \leq 0$ then 0
	else $(a - b) + K$

$a > b$	if $b - a < 0$ then 0
	else $(b - a) + K$

$a \geq b$	if $b - a \leq 0$ then 0
	else $(b - 1) + K$

$a \vee b$	min(fit(a), fit(b))
$a \wedge b$	fit(a) + fit(b)
$-a$	Negation is moved inwards and propagated over a

Table 3.1: Fitness function cited from (TCMM00).

constitute the search space. Search based techniques iteratively construct a set of test cases from such a space, attempting to maximise/minimise a predefined objective function. In most cases, the landscape of the input space is not known. This makes it very difficult to determine when to stop the search for the test data.

Currently, two rules are applied for terminating the computation. In the first case, a comparatively large number of iterations is defined, which leads to a saturated computation. However, if the computation converges at a very early stage, this termination rule will result in redundant computation. For example, if a predefined number of iterations is 1000 and the computation converges in 100 iterations, the rest of the computation contributes no effort in improving the quality of test data and becomes redundant. The other rule is to define a time period. If the performance of the exploration has no significant improvement within such a period, it is assumed that the maximum/minimum value is reached and the computation is terminated. This, however, can also lead to some problems. If the computation gets stuck in a local optimal within such a period, the process of search is terminated and the final result obtained is not globally optimal.

Chapter 4

Testing from finite state machines

4.1 Introduction

Finite state machines are formal specification languages. Since being developed, finite state machines have been used for modelling systems in various areas such as sequential circuits (Moo56; Hen64; KK68; Hsi71), software engineering (Cho78) and communication protocols (ATLU91; MP93; Hie96; SLD92; YU90).

In system development, a system is often modelled as a set of functional units or components. Some of these units are connected with others through input/output coupling. Each unit in the system is assigned with two attributes: an I/O port and an internal state. I/O port provides developers with an interface for the output observation when an input is sent while the internal state is not observable and can only be inferred from exhibited I/O behaviour.

Finite state machines, which involve a finite number of states and transitions between these states, are suitable for describing and implementing the control logic for applications. Compared to the other specification languages, finite state machines have several advantages: (1) they are comparatively simple, which makes it easy for inexperienced developers to implement; (2) in deterministic finite state machines, given a set of inputs and a known current state, the terminal state is predictable. The predicability of finite state machines helps to reduce the complexity of testing; (3) as a finite state machine can be represented by a directed graph, techniques involving directed graph theory can be applied with little modification; and (4) finite state machines have been studied for many years and become mature techniques for system modelling and testing.

Finite state machines have been considered as a useful way for automating the process of test generation from the system being developed. In finite state machine based testing, a test design is usually accomplished in two steps:

1. define a test strategy for the test objects;

2. automate the process of generating test sequences.

Test objects refer to all transitions contained in the finite state machine under test. A standard test strategy usually tests a transition in two parts: I/O behaviour check and the final state verification. Details about the standard test strategy are discussed in section 4.3. A complete testing sequence guarantees that all transitions contained in the finite state machine have been adequately tested. In the test process, it is usually preferred that a test sequence will continuously test all transitions without being mandatorily halted (say, manually reset). In section 4.4, the process that controls the generation of test sequences is introduced and in section 4.5, minimisation on the length of a test sequence is discussed.

4.2 Finite state machines

Definition 4.2.1 *A finite state machine (FSM) M is defined as a 5-tuple*

$$M = (I, O, S, \delta, \lambda, s_0)$$

where I, O, and S are finite and nonempty sets of input symbols, output symbols, and states, respectively; $\delta : S \times I \longrightarrow S$ is the state transition function; and $\lambda : S \times I \longrightarrow O$ is the output function. s_0 is the initial state of the machine.

In an FSM, if the machine receives an input $a \in I$ when in state $s_i \in S$, it moves to the state $s_{i+1} = \delta(s_i, a)$ and produces output $\lambda(s_i, a)$. Functions δ and λ can be extended to take input sequences in the usual way. An input sequence $x = a_1a_2...a_k$ takes the machine from a start state s_l successively to state $s_{i+1} = \delta(s_i, a_i)$, $i = l, ..., k$, with the final state $s_{k+1} = \delta(s_l, x)$, and produces an output sequence $b_1, ..., b_k = \lambda(s_l, x)$ where $b_i = \lambda(s_i, a_i)$, $i = l, ..., k$.

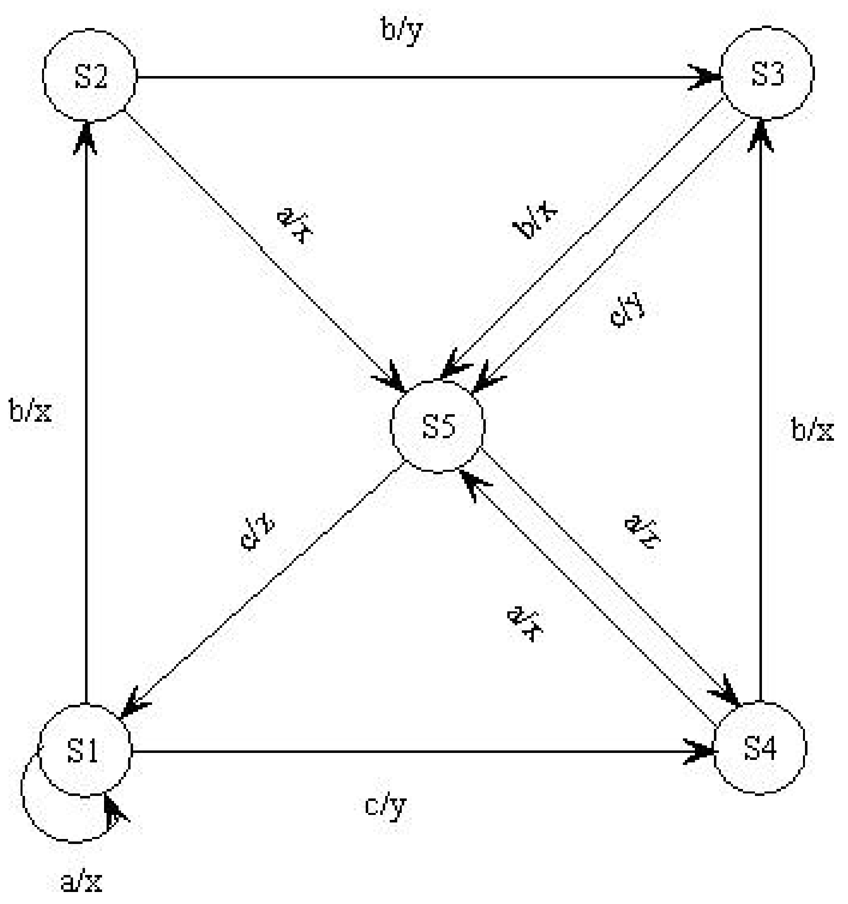


Figure 4.1: Finite state machine represented by digraph cited from ref. (ATLU91).

	a	b	c
s_1	s_1, x	s_2, x	s_4, y
s_2	s_5, x	s_3, y	None
s_3	None	s_5, x	s_5, y
s_4	s_5, x	s_3, x	None
s_5	s_4, z	None	s_1,z

Table 4.1: Finite state machine represented by state table.

An FSM M can be represented by a labelled directed graph $D = (V, E)$, where the set of vertices V represents the state set S of M and the set of edges E represents the transitions. An edge has label a/o where $a \in I$ and $o \in O$ are the corresponding transition's input and output. Figure 4.1 (copied from ref. (ATLU91)) illustrates an FSM represented by its corresponding directed graph. Alternatively, an FSM can be represented by a state table where one row lists each state and one column is used for each input. Table 4.1 shows the state table of the above FSM.

An FSM is *deterministic* if and only if for any state $s_i \in S$ and an input $a \in I$ there is at most one transition involving s_i and a. Two states s_i and s_j of M are said to be *equivalent* if and only if for every input sequence $\alpha \in I$ the machine produces the same output sequence, $\lambda(s_i, \alpha) = \lambda(s_j, \alpha)$. Machines M_1 and M_2 are *equivalent* if and only if for every state in M_1 there is an equivalent state in M_2, and vice versa. A machine M is *minimal* (*reduced*) if and only if no FSM with fewer states than M is equivalent to M.

It is assumed that any FSM being considered is minimal since any (deterministic) FSM can be converted into an equivalent (deterministic) minimal FSM (LY96). An FSM is *completely specified* if and only if for each state s_i and input a, there is a specified next state $s_{i+1} = \delta(s_i, a)$, and a specified output $o_i = \lambda(s_i, a)$; otherwise, the machine is *partially specified*.

A partially specified FSM can be converted to a completely specified one in two ways (LY96). One way is to define an error state. When a machine is in state s and receives an input a such that there is no transition from s with input a, it moves to the error state with a given (error) output. The other way is to add a loop transition. When receiving an undefined input, the state of a machine remains unchanged. At the same time, the machine produces no output. An FSM is *strongly connected* if the underlying directed graph is strongly connected.

It is assumed throughout this thesis that finite state machines under test are deterministic, minimal, completely specified and strongly connected.

4.3 Conformance testing

Given a specification FSM M, for which we have its complete transition diagram, and an implementation M', for which we can only observe its I/O behaviour ("black box testing"), we want to test to determine whether the I/O behaviour of M' conforms to that of M. This is called *conformance testing*. A test sequence that solves this problem is called a *checking sequence*.

An I/O difference between the specification and implementation can be caused by either an incorrect output (an output fault) or an earlier incorrect state transfer (a state transfer fault). Hennie (Hen64) suggests that the latter can be detected by adding a final state check after a transition check is finished. In Hennie's work, a test procedure is split into two parts, namely, I/O check and tail state verification. This contributes to the current standard test strategy. A standard test strategy is:

1. Homing: Move M' to an initial state s;
2. Output Check: Apply an input sequence α and compare the output sequences generated by M and M' separately;
3. Tail State Verification: Using state verification techniques to check the final state.

It is assumed throughout this thesis that all test sequences are generated using the standard test strategy.

The first step in the test is known as homing a machine to a desired initial state. This can be accomplished by using a homing sequence or a synchronizing sequence.

Definition 4.3.1 *A homing sequence H is an input sequence such that the output sequence on H uniquely determines the state reached after applying H.*

Definition 4.3.2 *A synchronizing sequence SN is an input sequence that moves the FSM to some fixed state irrespective of the initial state.*

Moore (Moo56) proved that all FSMs have homing sequences. However, not all FSMs have a synchronizing sequence. Kohavi (Koh78) proved that, if an FSM has a synchronizing sequence, the length of the sequence does not exceed $n(n-1)(n+1)/6$ where n is the number of states in the FSM. However, in his work, no algorithm is proposed for the construction of a synchronizing sequence. In case there exists no synchronizing sequence, an *adaptive synchronizing experiment* can be considered to move the FSM to a desired state.

Definition 4.3.3 *An adaptive synchronizing experiment is a strategy that will place the FSM in a particular state but with the input values used depending upon the previous output.*

Obviously, an adaptive synchronizing experiment is feasible in every strongly connected finite state machine as it is sufficient to execute a homing sequence to move the machine to a known state and then apply a transfer sequence to move the FSM to the desired state.

The second step in the test checks whether M' produces the desired output sequence. A fault detected at this stage is called an *output fault*. The last step checks whether M' is in the expected state $s' = \delta(s, \alpha)$ after the transition. A fault detected at this stage is called a *state transfer fault*. There are three main techniques used for state verification:

- Distinguishing Sequence (DS)
- Unique Input/Output (UIO)
- Characterizing Set (CS)

Definition 4.3.4 *A distinguishing sequence is an input sequence that produces a different output for each state.*

Execution of a DS provides evidence of the state an FSM is in when the DS is input. However, not every FSM has a DS (LY94).

Definition 4.3.5 *A UIO sequence of state s_i is an input/output sequence x/y, that may be observed from s_i, such that the output sequence produced by the machine in response to x from any other state is different from y, i.e. $\lambda(s_i, x) = y$ and $\lambda(s_i, x) \neq \lambda(s_j, x)$ for any $i \neq j$.*

A DS defines a UIO sequence for every state. While not every FSM has a UIO for each state, some FSMs without a DS have a UIO for each state.

Definition 4.3.6 *A characterizing set W is a set of input sequences with the property that, for every pair of state $(s_i,\ s_j)$, $j \neq i$, there is some $w \in W$ such that $\lambda(s_i, w) \neq \lambda(s_j, w)$. Thus, the output sequences produced by executing each $w \in W$ from s_j verifies s_j.*

The uses of DS, CS and UIO for state verification are proposed by Hennie (1964) (Hen64), Chow (1978) (Cho78) and Chen *et al.* (1989) (CVI89) separately. Compared to the other two techniques, the use of UIOs has several advantages: (1) Not all FSMs have a Distinguishing Sequence (DS), but nearly all FSMs have UIOs for each state (LY94); (2) The length of a UIO is no longer than that of a DS; (3) While UIOs may be longer than a characterizing set, in practice UIOs often lead to shorter test sequences.

However, computing a DS or UIOs from an FSM is NP-complete (LY94). Lee *et al.* (LY94) note that adaptive distinguishing sequences and UIOs may be produced by constructing a state splitting tree from the FSM being investigated.

A State Splitting Tree (SST) is a rooted tree T that is used to construct adaptive distinguishing sequences or UIOs from an FSM. Each node in the tree has a predecessor (parent) and successors (children). A tree starts from a root node and terminates at discrete partitions: sets that contain one state only. The predecessor of the root node, which contains the set of all states, is null. The nodes corresponding to a single state have empty successor. These nodes are also known as terminals. A child node is connected to its parent node through an edge labelled with characters. The edge implies that the set of states in the child node is partitioned from that in the parent node upon receiving the labelled characters. The splitting tree is complete if the partition is a discrete partition.

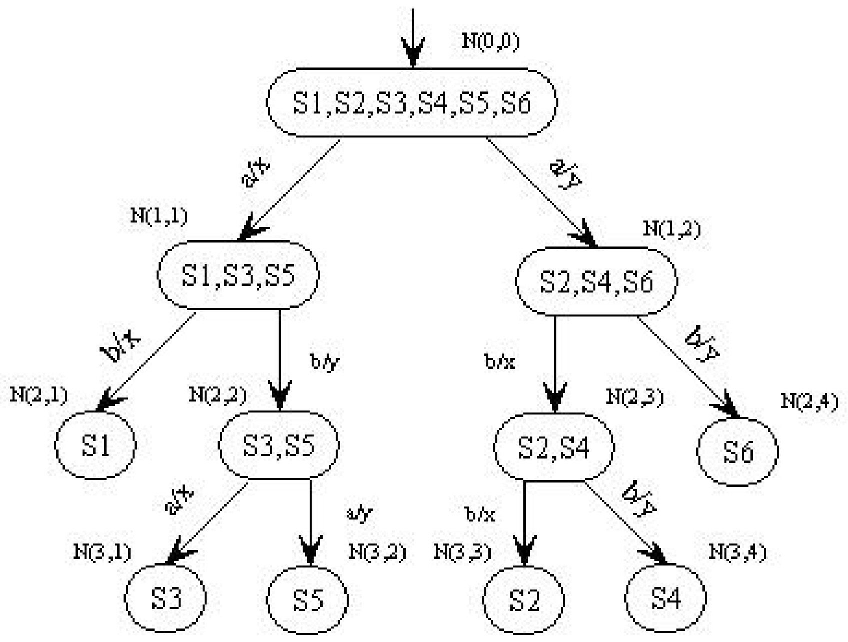


Figure 4.2: A pattern of state splitting tree from an FSM.

An example is illustrated in Figure 4.2 where an FSM (different from the one shown in Figure 4.1) has six states, namely, $S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The input set is $I = \{a, b\}$ while the output set is $O = \{x, y\}$. The root node is indicated by $N(0,0)$[1], containing the set of all states. Suppose states $\{s_1, s_3, s_5\}$ produce x when responding to a, while $\{s_2, s_4, s_6\}$ produce y. Then $\{s_1, s_3, s_5\}$ and $\{s_2, s_4, s_6\}$ are distinguished by a. Two new nodes rooted from $N(0,0)$ are then generated, indicated by $N(1,1)$ and $N(1,2)$. If we then apply b, the state reached from $\{s_1\}$ by a produces x while the states reached from $\{s_3, s_5\}$ by a produce y. Thus ab distinguish $\{s_1\}$ from $\{s_3, s_5\}$. Two new nodes rooted from $N(1,1)$ are generated, denoted by $N(2,1)$ and $N(2,2)$. The same operation can be applied to $\{s_2, s_4, s_6\}$. Repeating this process, we can get all discrete partitions as shown in Figure 2. Note that for some FSMs this process might terminate without producing a complete set of discrete partitions since there need not exist such a tree (LY96). A path from a discrete partition node to the root node forms a UIO for the state related to this node. When the splitting tree is complete, we can construct UIOs for each state.

Unfortunately, the problem of finding data to build up the state splitting tree is NP-hard. This provides the motivation for this PhD work, which investigates the use of MOTs for the construction of UIOs. The problem is discussed in chapter 5.

Shen *et al.* (SST91) shows that using a backward UIO (B-UIO) in a transition test helps to improve test quality. By applying a B-UIO, the initial state of the transition is also verified. All states in B-UIO method are verified twice. The test quality is therefore improved.

Definition 4.3.7 *A Backward UIO (B-UIO) sequence of state s_i is an input/output sequence x/y, that can be observed only if the final state of transitions is s_i, i.e. $\forall s_j\ (s_j \in S)$, $\lambda(s_j, x) = y \Rightarrow \delta(s_j, x) = s_i$.*

When both UIOs (forward) and B-UIOs are considered, the method is called the B-UIO method, or simply B-method. The test strategy is then:

[1] $N(i,j)$: i indicates that the node is in the ith layer from the tree. j refers to the jth node in the ith layer.

1. Homing: Move M' to the initial state of the B-UIO;
2. Initial state verification: Apply the B-UIO sequence to move M' to the initial state of a transition;
3. Output check: Apply an input α and compare the outputs generated by M and M' separately;
4. Tail state verification: Apply UIO (forward) to check the final state.

The UIO (forward) is used to provide confidence that the tested transition arrives at a correct state while the Backward UIO is used to increase the confidence that an assigned transition has been tested.

4.4 Test sequence generation

When testing from FSMs, it is always desirable that a test sequence will detect all faults from the Implementation Under Test (IUT) with the shortest length. In finite state machine based testing, testing is performed in "black-box" manner. Thus, the number of states in the IUT is not known. This requires some assumptions before testing starts.

It has been shown that, if the number of states in the IUT exceeds the number of states in its specification, the process of conformance testing is NP-hard (Moo56). Moore (Moo56) proved that, given a minimal FSM M and a faulty implementation $M^{'}$, it is always possible to generate an input sequence that will distinguish $M^{'}$ from M as long as $M^{'}$ has no more states than M. Based upon Moore's work, four test methods are proposed for the generation of test sequences from the FSM under test. They are T-method (NT81), D-method (Hen64), W-method (Cho78) and U-method (CVI89). T-, D- and U-methods assume that the number of states in the IUT does not exceed that in the specification FSM. W-method allows more states in the IUT but assumes that the maximum number of states that the correct design might have is known.

In the T-method, a test sequence (called a transition-tour sequence) is generated simply by applying inputs to the specification FSM until all transitions in the FSM have been traversed at least once. In the D-, W-, and U-methods, as well as each transition being executed, tail state of the transition needs to be verified by DS, CS, and UIO correspondingly. D-, W-, and U-methods are called formal methods. In terms of fault coverage, D-, W-, and U-methods outperform T-method (SL89).

A method for generating a complete test sequence is described as follows: an untested transition is selected from the specification FSM first. The FSM is then moved to the initial state of the selected transition by using a transfer sequence. This transfer sequence is also called a *linking sequence* in the final test sequence. Normally, the test process starts with a transition whose starting state is the initial state of the FSM where no transferring operation is required. The selected transition is then tested by checking its I/O plus verifying the tail state. An input sequence that will test the selected transition is called the *test sequence* for this transition.

After a test is complete, another transition is selected. The FSM is then moved to the initial state of the selected transition by a transfer sequence. The process of transition testing starts and the selected transition is tested by its test sequence. The process repeats until all transitions have been adequately tested.

However, test sequence generated by such a process might result in a long sequence, which consequently increases the cost in the forthcoming implementation test. A short test sequence is thus preferred. This leads to the problem of minimisation on the length of a test sequence.

Clearly, the increment of a test sequence's length is caused by the uses of linking sequences. If, when generating a test sequence, a set of linking sequences is applied such that the sum of the length of all linking sequences is minimal, the test sequence is of the minimal length. In the next section, techniques for generating an optimal test sequence from an FSM are discussed.

4.5 Optimisation on the length of test sequences

4.5.1 Single UIO based optimisation

Aho *et al.* (ATLU91) noted that an optimal test sequence may be produced when UIOs are used for state verification. In their work, Aho *et al.* proved that, if an

FSM M is strongly connected or weakly connected with all states having reset functions, there exists a Chinese postman tour T in the test control digraph of M. Inputs derived from T constitute an optimal test sequence for M. Based upon rural Chinese postman tour algorithm, an optimal technique for the generation of a test sequence from the FSM under test is proposed. This technique is described as follows.

Let us suppose that we have a specification FSM M with the input set I and the output set O respectively. The FSM is represented by a digraph $D = (V, E)$ where V represents the set of states and E the set of transitions. Let $(v_i, v_j, a_m/o_n) \in E$ be a transition under test where v_i is the initial state and v_j the final state. a_m/o_n is the I/O pair of this transition, $a_m \in I, o_n \in O$. Let UIO_j be the UIO sequence for v_j. When in v_j and receiving the input part of UIO_j, M produces the output part of UIO_j and arrives at v_k. The final state of UIO_{s_j} is denoted as $v_k = Tail(UIO_j)$. The test sequence for a transition $(v_i, v_j, a_m/o_n)$ is constructed by concatenating a_m with the input part of UIO_j. After the test of $(v_i, v_j, a_m/o_n)$ is complete, a *test control trail*[1] for the transition is produced, denoted as [2]$(v_i, v_k; (a_m/o_n) \cdot UIO_j)$, $Tail(UIO_j) = v_k$. If all transitions in M have been tested by their test sequences, we can finally get a complete set of test control trails. This set is denoted as:

$$E_C = \{(v_i, v_k; (a_m/o_n) \cdot UIO_j) : (v_i, v_j; a_m/o_n) \in E \ \wedge \ Tail(UIO_j) = v_k\}.$$

A digraph $D^{'} = (V^{'}, E^{'})$ is then constructed such that $V^{'} \equiv V$ and $E^{'} = E_C \cup E$. Digraph $D^{'}$ is called the *test control digraph* of D. It is easy to see that if an input sequence traverses all edges in $D^{'}$ at least once, then all transitions in D (or more precisely M) are tested at least once. The problem of generating an optimal test sequence from D is now converted to that of finding the Chinese postman tour in $D^{'}$. As introduced in chapter 2, if $D^{'}$ is symmetric, there exists an Euler tour in $D^{'}$ and the Euler tour is also a Chinese postman tour; otherwise,

[1]A test control trail implies the sequence of transitions that the machine will traverse when a transition is tested by its test sequence. Here, only the initial state of the first transition and the final state of the last transition are drawn.

[2]The notation '$\cdot$' indicates the concatenation of two sequences.

State	UIO	Final State
v_1	ba/xx	v_5
v_2	b/y	v_3
v_3	bc/xz	v_1
v_4	bc/xy	v_5
v_5	c/z	v_1

Table 4.2: UIO sequences for the states of the FSM shown in fig.4.1.

$D^{'}$ needs to be augmented to derive its symmetric augmentation digraph D^{*}. Once D^{*} is complete, the Euler tour in D^{*} is equivalent to the Chinese postman tour in $D^{'}$. When augmenting $D^{'}$, the cost of an edge in $D^{'}$ is calculated as:

$$C(v_i, v_k; (a_m/o_n) \cdot UIO_j) = C(v_i, v_j; (a_m/o_n)) + C(UIO_j).$$

A flow network is thus required for calculating the replication times $\psi(v_i, v_j; a_m/o_n)$ for a transition $(v_i, v_j; a_m/o_n) \in E$.

To further explain the control scheme, an example is illustrated where the FSM shown in Figure 4.1 is utilised. The UIOs for states are listed in table 4.2. We also allow the FSM to have a reset function ri. When receiving ri, the machine returns to v_1 and produces $null$, regardless of the current state.

In order to generate the test control digraph $D^{'}$ for the FSM, each transition's test control trail needs to be derived. For example, the test control trail for transition $(s_1, s_1; a/x)$ is constructed by $(v_1, v_5; (a/x) \cdot (ba/xx)), Tail(UIO_1) = Tail(ba/xx) = v_5$. An edge starting with v_1, ending with v_5 and labelled with aba/xxx is added to digraph $D^{'}$. The test cost for this edge is calculated as $C(v_1, v_5; (a/x) \cdot (ba/xx)) = C(v_1, v_1; a/x) + C(v_1, v_5; ba/xx)$.

Let the cost of an edge in the FSM be 1. Test cost of edge $(v_1, v_5; (a/x) \cdot (ba/xx)), Tail(UIO_1) = Tail(ba/xx) = v_5$ is then 3. There are 11 transitions in the FSM. All transition's test control trails need to be added to $D^{'}$. Test control trails of all transitions are listed in table 4.3. Five additional test control trails that are related to reset function are also included in the final test control digraph. The final test control digraph generated from table 4.3 and Figure 4.1 is shown in Figure 4.3.

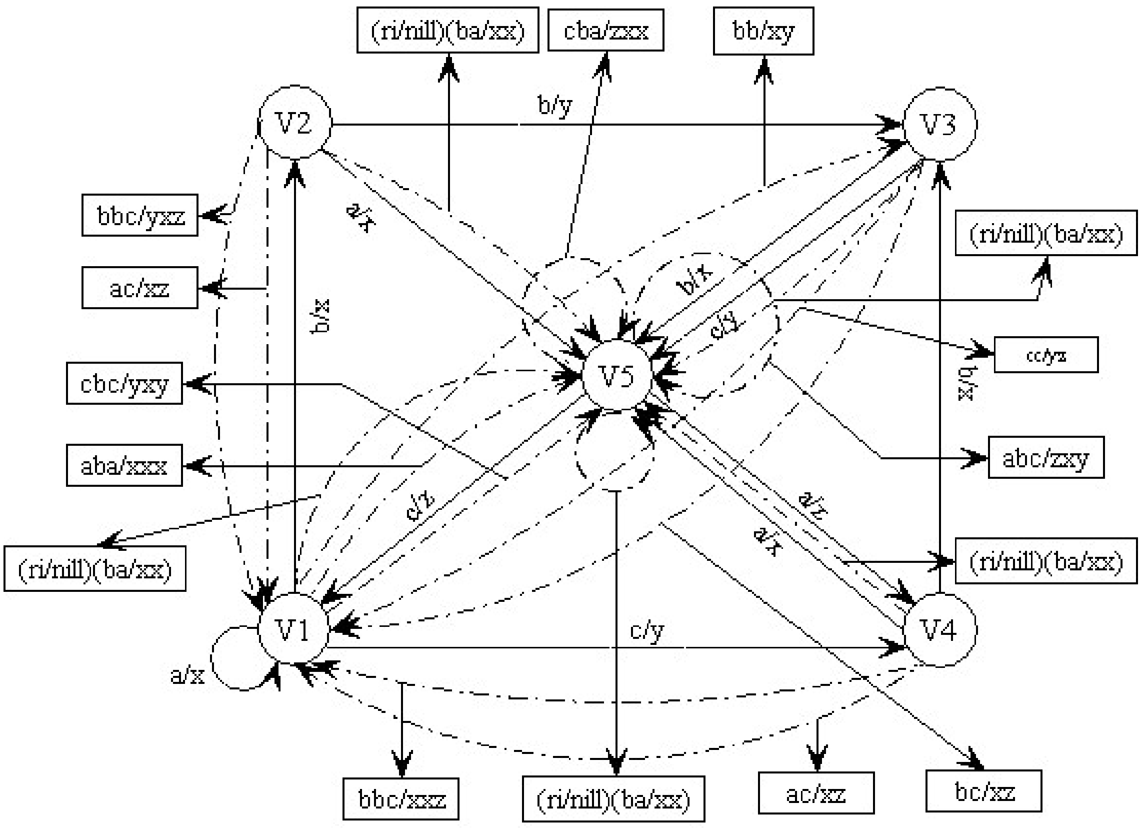


Figure 4.3: Test control digraph of the FSM shown in fig. 4.1 using single UIO for each state.

No	Transition	Tail state verification	Test control trail
1	$(v_1, v_1; a/x)$	$(v_1, v_5; ba/xx)$	$(v_1, v_5; aba/xxx)$
2	$(v_1, v_4; c/y)$	$(v_4, v_5; bc/xy)$	$(v_1, v_5; cbc/yxy)$
3	$(v_1, v_2; b/x)$	$(v_2, v_3; b/y)$	$(v_1, v_3; bb/xy)$
4	$(v_2, v_3; b/y)$	$(v_3, vs_1; bc/xz)$	$(v_2, v_1; bbc/yxz)$
5	$(v_2, v_5; a/x)$	$(v_5, v_1; c/z)$	$(v_2, v_1; ac/xz)$
6	$(v_3, v_5; b/x)$	$(v_5, v_1; c/z)$	$(v_3, v_1; bc/xz)$
7	$(v_3, v_5; c/y)$	$(v_5, v_1; c/z)$	$(v_3, v_1; cc/yz)$
8	$(v_4, v_3; b/x)$	$(v_3, v_1; bc/xz)$	$(v_4, v_1; bbc/xxz)$
9	$(v_4, v_5; a/x)$	$(v_5, v_1; c/z)$	$(v_4, v_1; ac/xz)$
10	$(v_5, v_1; c/z)$	$(v_1, v_5; ba/xx)$	$(v_5, v_5; cba/zxx)$
11	$(v_5, v_4; a/z)$	$(v_4, v_5; bc/xy)$	$(v_5, v_5; abc/zxy)$
12	$(v_1, v_1; ri/null)$	$(v_1, v_5; ba/xx)$	$(v_1, v_5; (ri/null)(ba/xx))$
13	$(v_2, v_1; ri/null)$	$(v_1, v_5; ba/xx)$	$(v_2, v_5; (ri/null)(ba/xx))$
14	$(v_3, v_1; ri/null)$	$(v_1, v_5; ba/xx)$	$(v_3, v_5; (ri/null)(ba/xx))$
15	$(v_4, v_1; ri/null)$	$(v_1, v_5; ba/xx)$	$(v_4, v_5; (ri/null)(ba/xx))$
16	$(v_5, v_1; ri/null)$	$(v_1, v_5; ba/xx)$	$(v_5, v_5; (ri/null)(ba/xx))$

Table 4.3: Test control trails of transitions in the FSM shown in fig. 4.1.

The problem now comes to finding the Chinese postman tour in D'. As one can see D' is not symmetric and therefore does not contain an Euler tour. The symmetric augmentation of D' is thus required. In order to construct the symmetric augmentation digraph D^* of D', a flow network $D_F = (V_F, E_F)$ needs to be constructed. D_F is defined as $V_F = V(D') \cup \{s, t\}$ where s is the source and t the sink, and $E_F = E(D) \cup E^+ \cup E^-$ where $E^+ = \{(s, v_1), (s, v_5)\}$, $u(s, v_1) = 2$, [1]$u(s, v_5) = 6$; $E^- = \{(v_2, t), (v_3, t), (v_4, t)\}$, $u(v_2, t) = 3$, $u(v_3, t) = 2$, $u(v_4, t) = 3$. E^+ and E^- are derived through calculating the index of each vertex (state), $\xi(v_i) = d^-_{v_i} - d^+_{v_i}$. Edges in E^+ and E^- have zero test cost. The capacity of these edges are defined as $u(v_i, s) = \xi(v_i)$, $u(t, v_j) = -\xi(v_j)$. The rest of the edges in E' have the same cost as those edges in E and have infinite capacities (for more details, see chapter 2).

Thus, in terms of a max flow F, a function $\psi(v_i, v_j; a_m/o_n)$ is required to determine the times of replication for the edge $(v_i, v_j) \in E$ in D^*. The function

[1]$u(s, v_i)$ defines the capacity of edge $(s,\ v_i)$.

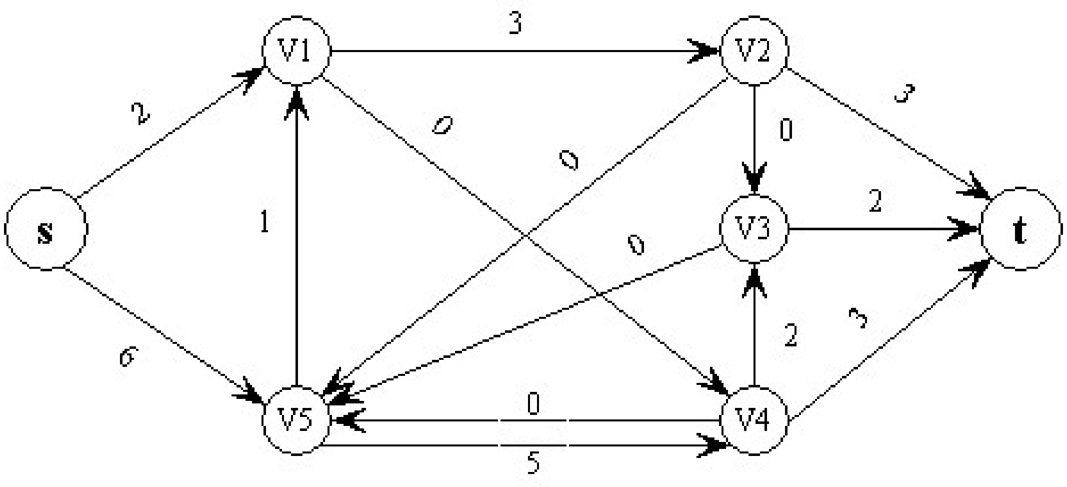


Figure 4.4: The flow graph D_F of the digraph D' with the maximum flow and minimum cost.

is defined as:

$$\psi(v_i, v_j; a_m/o_n) = \begin{cases} 1, & \text{if } (v_i, v_j; a_m/o_n) \in E_C \\ F(v_i, v_j; a_m/o_n), & \text{if } (v_i, v_j; a_m/o_n) \in E \end{cases}$$

Finding a maximum-flow/minimum-cost flow F on D_F over ψ leads to a symmetric augmentation D^* of D'(ATLU91). F gives us ψ which defines the times of replication of edge (v_i, v_j) in D^*. The final flow in the network is shown in Figure 4.4. The number on each edge indicates the times that this edge needs to be replicated.

Having replicated edges in D' by the times given in D_F, a symmetric augmentation digraph D^* is generated shown in Figure 4.5. An Euler tour can then be constructed from D^* by using the algorithm proposed by (Kua62). This Euler tour is also a Chinese postman tour in D'. Test sequence derived from this tour is an optimal test sequence for the FSM shown in Figure 4.1. The result is shown in table 4.4. There are total 55 input/output pairs.

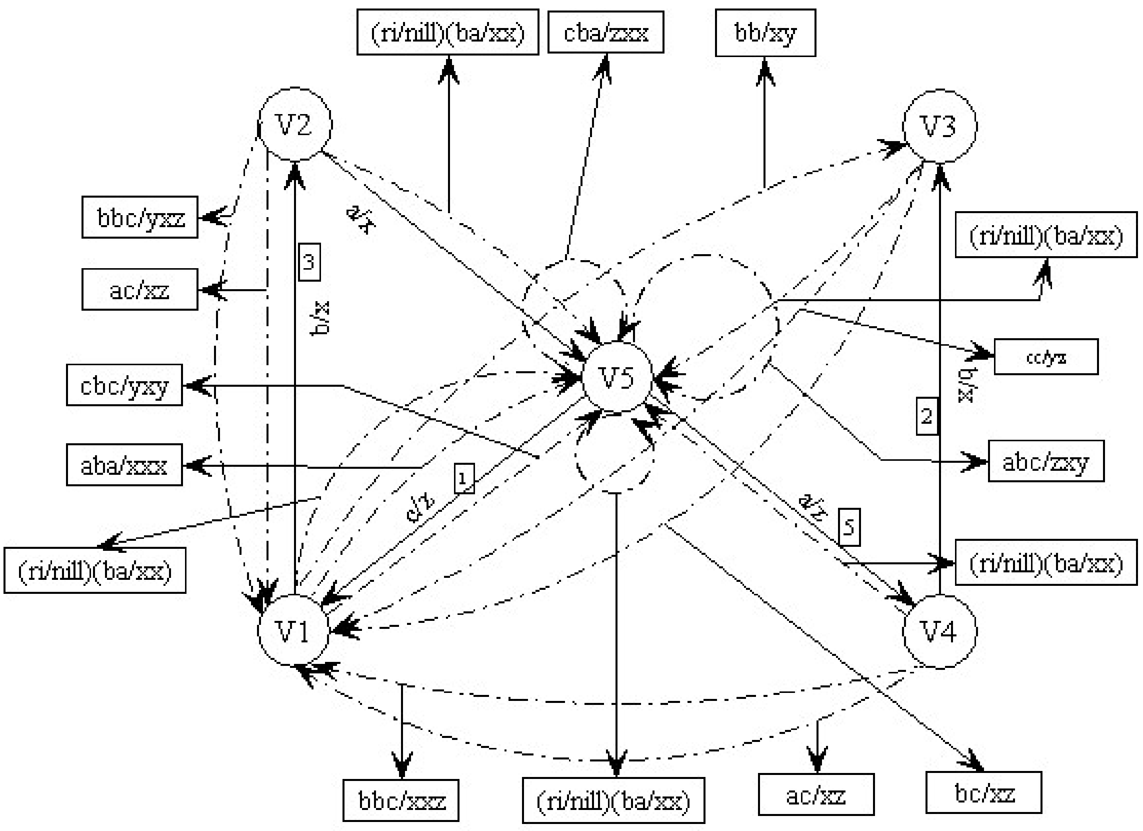


Figure 4.5: Symmetric augmentation from fig. 4.3. Number in an edge indicates the times this edge needs to be replicated.

$ri/null$	bb/xy	cc/yz	aba/xxx
abc/zxy	cba/zxx	$(ri/null) \cdot (ba/xx)$	a/x
ac/xz	cbc/yxy	a/z	b/x
bc/cz	b/x	bbc/yxz	b/x
ac/xz	b/x	$(ri/null) \cdot (ba/xx)$	a/z
b/x	$(ri/null) \cdot (ba/xx)$	a/z	bbc/xxz
$(ri/null) \cdot (ba/xx)$	a/z	$(ri/null) \cdot (ba/xx)$	c/z

Table 4.4: An optimal test sequence for the FSM shown in fig. 4.1 by using single UIO for each state.

Index	State	UIO	Final State
UIO_1^1		aa/xx	v_1
UIO_1^2		ab/xx	v_2
UIO_1^3	v_1	cb/yx	v_3
UIO_1^4		ac/xy	v_4
UIO_1^5		ba/xx	v_5
UIO_2^1	v_2	b/y	v_3
UIO_3^1	v_3	bc/xz	v_1
UIO_3^2		ba/xz	v_4
UIO_4^1	v_4	bc/xy	v_5
UIO_5^1	v_5	c/z	v_1
UIO_5^2		a/z	v_4

Table 4.5: Multiple UIO sequences for the states of the FSM shown in fig.4.1.

$ri/null \quad aaa/xxx \quad bb/xy \quad bc/xz \quad ri/null \quad ab/xx$
$ri/null \quad ab/xx \quad ac/xz \quad cb/yx \quad c/y \quad abc/zxy$
$ri/null \quad ba/xx \quad ccb/zyx \quad (ri/null) \cdot (ab/xx)$
$b/y \quad ba/xz \quad aa/xz \quad bba/xxx \quad (ri/null) \cdot (cb/yx)$
cc/yz

Table 4.6: The optimal test sequence for the FSM shown in fig. 4.1 by using multiple UIOs for state verification.

4.5.2 Multiple UIOs based optimisation

Shen *et al.* (SLD92) note that the length of a test sequence can be further reduced if multiple UIOs are applied for each state. In the example shown in Figure 4.1, there exits more than one UIOs for each state. These UIOs are listed in table 4.5. A test control digraph, generated by using different UIOs for a state verification, is shown in Figure 4.6. From the graph it can be seen that $\xi(v_1) = \xi(v_2) = \xi(v_3) = \xi(v_4) = \xi(v_5) = 0$. The digraph is symmetric and needs no augmentation. The corresponding test sequence constructed from the test control digraph is shown in table 4.6. There are 44 input/output pairs, ll shorter than the one constructed using single UIO for each state.

Shen *et al.*'s method is motivated from the observation that the number of times for replicating edges in an FSM M is determined by the indexes of all

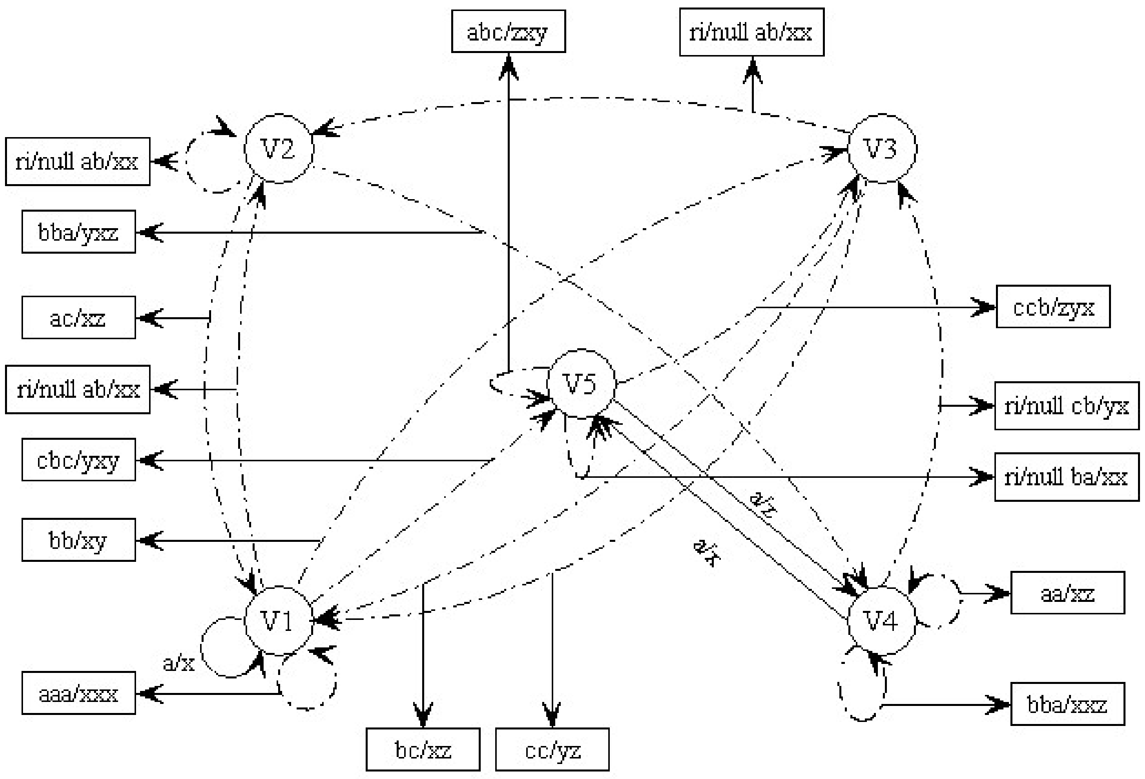
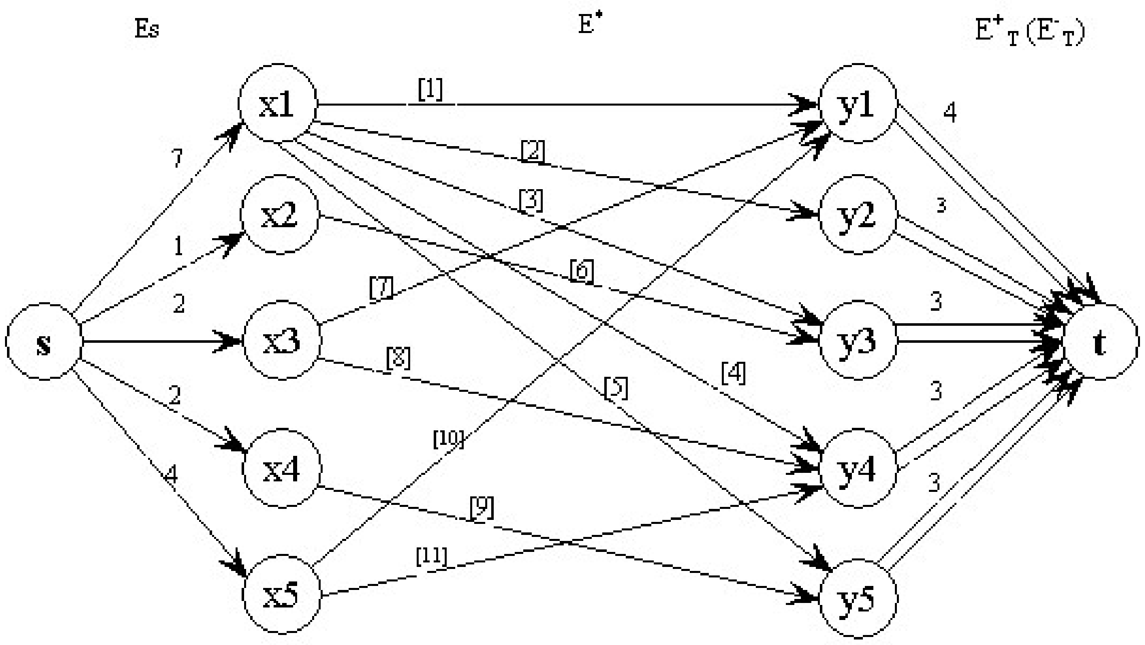


Figure 4.6: Test control digraph of the FSM shown in fig. 4.1 using multiple UIOs for each state.



[1]. UIO_1^1 [2]. UIO_1^1 [3]. UIO_1^1 [4]. UIO_1^1 [5]. UIO_1^1

[6]. UIO_2^1

[7]. UIO_3^1 [8]. UIO_3^2

[9]. UIO_4^1

[10]. UIO_5^1 [11]. UIO_5^2

Figure 4.7: Flow network used for the selection of UIOs to generate optimal test control digraph for the FSM shown in fig. 4.1.

vertices (states) in the test control digraph D'. If, for all vertices in D', there exist $\xi(v_i) = 0$, $i = 1, ..., n$, then the test control digraph is symmetric and no replication is required. The cost of the test sequence is simply the cost of the edges in E_C. In Aho *et al.*'s method, only one UIO is applied for each state. The index $\xi(v_i), i = 1, ..., n$, is thus fixed and is usually not equal to zero. This requires the symmetric augmentation by replicating some edges for a number of times, which consequently increases the length of the test sequence. It shall be noted that for a vertex v_i, its out-degree $d^-(v_i)$ in the test control digraph D' is always the same as that in D, but its in-degree $d^+(v_i)$ may vary if different UIOs are selected for a state when generating the test control digraph. Appropriate selection of a UIO for a transition test will lead to a minimal value of $\sum_{i=1}^n |\xi(v_i)|$ in D', which determines that the times for replicating edges in D is minimal. The problem is defined as follows:

Given an FSM M represented by digraph D and a set of UIOs, $MUIO_j = \{UIO_j^1, ..., UIO_j^r\}$, for each state $s_j \in S$, find an element $UIO_j^\alpha \in MUIO_j$ for transition $(v_i, v_j; a_m/o_n)$ such that, in the final test control digraph D', $\sum_{i=1}^n |\xi(v_i)|$ is minimal. Selection of a UIO for a transition test can be determined by constructing a network D_M and then finding a maximum-flow/minimum-cost flow F on D_M. The technique is described as follows:

Given M represented by D, D_M is defined as $V_M = X \times Y \times \{s, t\}$ where $X = \{x_1, ..., x_n\}$, $Y = \{y_1, ..., y_n\}$ and $X \equiv Y \equiv V(D)$[1]; s is the source and t the sink; $E_M = E_S \cup E_T^+ \cup E_T^- \cup E^*$ where $E_S = \{(s, x_i), \forall x_i \in X\}$, $E_T^- = \{(y_j, t), \forall y_j \in Y\}$, $E_T^+ = \{(y_k, t), \forall y_k \in Y\}$ and $E^* = \{(x_i, y_j; UIO_i), \forall\ UIO_i \in MUIO_i, Tail(UIO_i) = s_j\}$; each edge $(s, x_i) \in E_S$ has zero cost and capacity $u(s, x_i)$=$d^+(v_i)$; for each $y_j \in Y$, two edges are used to link y_j with t, one of which has a positive cost, while, the other has a negative cost; each edge $(y_j, t) \in E_T^-$ has cost -1 and capacity $u(y_j, t)$=$d^-(v_j)$; each edge $(y_j, t) \in E_T^+$ has +1 cost and infinite capacity; each edge $(x_i, y_i) \in E^*$ has zero cost and infinite capacity. A flow F_M on D_M is defined as a function $F : E_M \rightarrow Z^+$ such that:

1. $\forall x_i \in X$, $F(s, x_i) = \sum_{(x_i, y_j) \in E^*} F(x_i, y_j)$;

2. $\forall y_j \in Y$, $F(y_j, t)^+ + F(y_j, t)^-$= $\sum_{(x_i, y_j) \in E^*} F(x_i, y_j)$;

[1] X and Y are two complete sets of states denoted with different notations.

3. $\forall(s, x_i) \in E_S$, $F(s, x_i) \leq u(s, x_i)$;

4. $\forall(y_j, t) \in E_T^-$, $F(y_j, t) \leq u(y_j, t)$.

The cost of the flow is defined as: $C(F) = \sum_{(y_j,t)\in E_T^+} F(y_j, t)$ - $\sum_{(y_j,t)\in E_T^-} F(y_j, t)$.

Given a maximum-flow/minimum-cost flow F on G_M, the flow number on an edge $(x_i, y_j; UIO_i)$ indicates the times that UIO_i needs to be included in the test control digraph D'.

Theorem 4.5.1 *(SLD92) If a flow F on D_M is a maximum-flow/minimum-cost flow, then the corresponding assignment of UIO sequences to the edges of D is such that $\sum_{i=1}^{n} |\xi(v_i)|$ is minimal.*

D_M for the FSM shown in Figure 4.1 is illustrated in Figure 4.7.

4.5.3 Optimisation with overlap

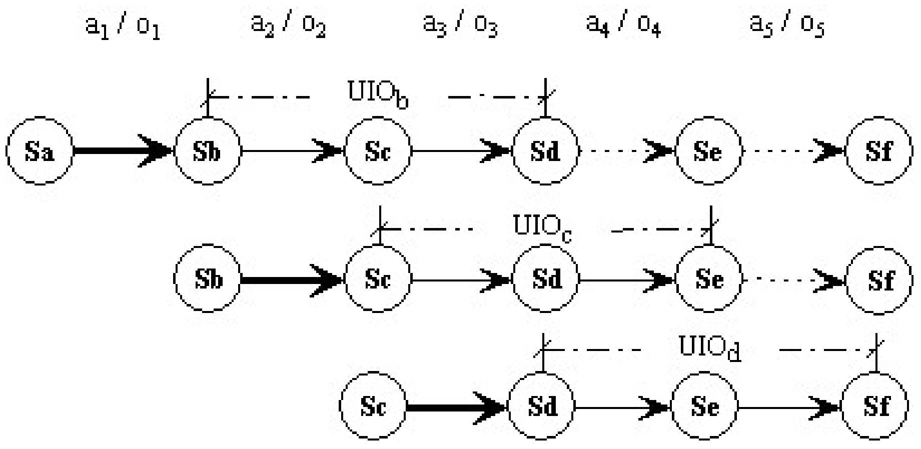


Figure 4.8: Structure of UIO Overlap.

In the methods proposed by Aho *et al.* and Shen *et al.*, a test control trail is generated simply for the use of one transition test. In order that all transitions are adequately tested, a complete set of test control trails for all transitions in E is required, each element in the set being independent from others. However, a test sequence for a transition test might contain a subsequence that is part of a test sequence for another transition test. The structure of two test sequences *overlap*.

Figure 4.8 illustrates an example. In the figure, a transition $(s_a, s_b; a_1/o_1)$ is tested by test sequence $TS_1 = (a_1/o_1) \cdot (a_2/o_2) \cdot (a_3/o_3)$. It can be noted that sequence $(a_2/o_2) \cdot (a_3/o_3)$ is also a part of test sequence TS_2, $TS_2 = (a_2/o_2) \cdot (a_3/o_3) \cdot (a_4/o_4)$, that tests transition $(s_b, s_c; a_2/o_2)$. Sequence $(a_2/o_2) \cdot (a_3/o_3)$ is the overlapped segment between TS_1 and TS_2. Test sequence $TS_3 = (a_1/o_1) \cdot (a_2/o_2) \cdot (a_3/o_3)$ contains overlapped segments with TS_1, (a_3/o_3), and TS_2, $(a_3/o_3) \cdot (a_4/o_4) \cdot (a_5/o_5)$ as well; if being applied to the FSM, test sequence $(a_1/o_1) \cdot (a_2/o_2) \cdot (a_3/o_3) \cdot (a_4/o_4) \cdot (a_5/o_5)$ will test transitions $(s_a, s_b; a_1/o_1)$ $(s_b, s_c; a_2/o_2)$ and $(s_b, s_c; a_3/o_3)$ successively.

Clearly, a test sequence generated by considering overlap among test sequences for a set of ordered and distinct transitions is shorter than that generated by concatenating test sequences that individually test each transition in the set. If, in a test sequence for the FSM under test, the overall overlap among test sequences for all transitions in E is maximised, the test sequence is of the minimal length. This leads to the problem of finding a set of ordered transitions such that overlap formed among the corresponding test sequences are maximised.

Yang *et al.* (YU90) and Miller (MP93) show that overlap can be used in conjunction with multiple UIOs to further reduce the test sequence length. Hierons (Hie96; Hie97) represents overlap by invertible sequence. All of their work aims to generate a set of fully overlapped transition sequences, each of which is used to generate a test control trail in the test control digraph. The test sequence derived from such a test control digraph is of the minimal length.

Definition 4.5.1 *A fully overlapped transition sequence (FOTS) is a sequence of distinct transitions such that if it is followed by a state identification sequence for the end state of the last transition in the sequence (so this transition is verified), then all other transitions in the sequence are also verified.*

Definition 4.5.2 *A transition $(s_i, s_j; a_m/o_n) \in E$ is an invertible transition if and only if it is the only transition entering state s_j that involves input a_m and output o_n.*

Definition 4.5.3 *A sequence of transitions* $t = t_1 ... t_n$*, with* $t_i = (s_i, s_j; a_{m_i}/o_{n_i})$*,* $a_{m_i} \in I, o_{n_i} \in O$*, is an Invertible Sequence (IS) if* s_i *is the only state that produces output sequence* $o_{n_1} ... o_{n_n}$ *and moves to state* s_j *when input* $a_{m_1} ... a_{m_n}$ *is applied.*

4.6 Other finite state models

Except for ordinary finite state machines, some other finite state models have been proposed for system modelling and testing, including Extended Finite State Machines (EFSMs), Communicating Finite State Machines (CFSMs) and Probabilistic Finite State Machines (PFSMs).

Definition 4.6.1 *An extended finite state machine (EFSM) EM is a quintuple*

$$EM = (I, O, S, \overrightarrow{x}, T)$$

where I*,* O*,* S*,* $\overrightarrow{x}$ *and* T *are finite sets of input symbols, output symbols, states, variables, and transitions, respectively. Each transition* t *in the set* T *is a six-tuple*

$$t = (s_t, q_t, a_t, P_t, A_t)$$

where s_t*,* q_t *and* a_t *are the start (current) state, end (next) state, input, and output, respectively.* $P_t(\overrightarrow{x})$ *is a predicate on the current variable values and* $A_t(\overrightarrow{x})$ *gives an action on variable values.*

Definition 4.6.2 *A communicating finite state machine (CFSM) CM is a 7-tuple*

$$CM = (I, O, S_i, A_i, \delta_i, M_i, s_{0_i})$$

where I *and* O *are the input set and output set respectively.* S_i *is the set of local states of machine* M_i*,* $s_{0_i} \in S_i$ *is the initial state of machine* M_i*;* $A_i = \cup_{1 \leq j \leq n} A_{i,j}$ $\bigcup \cup_{1 \leq j \leq n} A_{j,i}$ *where* $A_{i,j}, 1 \leq j \leq n$ *is the alphabet of messages that* M_i *can send to* M_j*, and* $A_{j,i}, 1 \leq j \leq n$ *is the alphabet of messages that* M_i *can receive from* M_j*.* $\delta_i : S_i \times A_i \times I \rightarrow S_i$ *is the transition function.* $\delta_i(p, -m, j)$ *is the set of states that process* M_i *could move to from state* p *after sending a message* m *to process* M_j*.* $\delta_i(p, +m, j)$ *is the set of states that process* M_i *could move to from state* p *after receiving a message* m *sent by process* M_j*.*

Obviously, a CFSM is equivalent to an EFSM. One can simply add a variable to encode the component machines and represent the CFSM by an EFSM. An EFSM with finite variable domains is a compact representation of an FSM (LY96). Therefore, the process of testing an EFSMs can be reduced to that of testing an ordinary FSMs by expanding the EFSM into an equivalent FSM. Once the conversion is complete, an optimal test sequence can be generated using the methods discussed above.

Definition 4.6.3 *A probabilistic finite state machine (PFSM) PM is a quintuple*

$$PM = (I, O, S, T, P)$$

where I, O, S, T are the input set, output set, state set, and transition set, respectively. Each transition $t \in T$ is a tuple $t = (s_i, s_j, a, o)$ consisting of the start (current) state $s_i \in S$, next state $s_j \in S$, input symbol $a \in I$ and output symbol $o \in O$. P is a function that assigns a number $P(t) \in [0, 1]$ to each transition t (its probability), so that for every state s and input symbol a, $\sum_{s_j,o} P(s, s_j, a, o) = 1$.

A PFSM can be represented by a transition graph with n nodes corresponding to the n states and directed edges between states corresponding to the transition with nonzero probability. Specifically, if $P(s_i, s_j; a/o) > 0$, then there is an edge from s_i to s_j with an associated input a and output o; otherwise, there is no corresponding edge.

Conformance testing for PFSM is to check whether an implementation PFSM confirms to a specification PFSM where the term of "conformance" can be defined in different ways (LY96). Conformance test for PFSM has not been extensively studied and the study of PFSM remains an open research topic (LY96).

Chapter 5

Construction of UIOs

5.1 Introduction

Unique Input/Output (UIO) sequences are often applied for state verification in finite state machine based testing. A UIO sequence for state s_i in a finite state machine M is an input/output sequence x/y, that may be observed from s_i, such that the output sequence produced by the machine in response to x from any other state is different from y. A prerequisite for UIO-based techniques is that there exists at least one UIO sequence for each state in the finite state machine under test. Thus, finding ways to (effectively) construct UIOs from a finite state machine is very important.

Unfortunately, computing UIOs is NP-hard (LY94). Lee and Yannakakis (LY94) note that adaptive distinguishing sequences and UIOs may be produced by constructing a state splitting tree[1]. However, no rule is explicitly defined to guide the construction of an input sequence. Naik (Nai97) proposes an approach to construct UIOs by introducing a set of inference rules. Some minimal length UIOs are found. These are used to deduce some other states' UIOs. A state's UIO is produced by concatenating a sequence to another state, whose UIO has been found, with this state's UIO sequence. Although it may reduce the time taken to find some UIOs, the inference rule inevitably increases a UIO's length, which consequently leads to longer test sequences.

[1]For more about state splitting tree, see chapter 4.

Metaheuristic Optimisation Techniques (MOTs) such as Genetic Algorithms (GAs) (Gol89) and Simulated Annealing (SA) (KGJV83; MRR+53) have proven efficient in search and optimisation and have shown their effectiveness in providing good solutions to some NP–hard problems such as the *Travelling Salesman Problem* (CCPS98). When searching for optimal solutions in multi-modal functions, the use of *sharing* techniques is likely to lead to a population that contains several sub-populations that cover local optima (GR87). This result is useful since in some search problems we wish to locate not only global optima, but also local optima.

In this chapter, a model is proposed for the construction of UIOs by using MOTs. A fitness function is defined to guide the search of input sequences that constitute UIOs for some states. The fitness function works by encouraging the early occurrence of discrete partitions in the state splitting tree constructed by an input sequence while punishing the length of this input sequence.

The study of the proposed model consists of two stages. In the first stage, the performance of GA on the construction of UIOs was initially investigated. Two simple FSMs were used for the experiments. The experiments were designed only to compare the performance between GAs and random search; in the second stage, a more thorough study was carried on where sharing techniques are applied. The use of a sharing technique forces the population of the genetic pool to form sub-populations, each of which aims to explore UIOs that are calculated as local optima. This helps to maintain the diversity of the genetic pool. A set of experiments is designed to study the performance of GAs, GAs with sharing, SA and SA with sharing respectively. The related work is discussed in the following sections.

5.2 Constructing UIOs with MOTs

5.2.1 Solution representation

When applying MOTs to a finite state machine, the first question that has to be considered is what representation is suitable. In this work, the potential solutions in a genetic pool are defined as strings of characters from the input set I. A ***DO***

NOT CARE character $'\sharp'$ is also used to further maintain diversity (this is explained in section 5.4). When receiving this input, the state of a finite state machine remains unchanged and no output is produced. When a solution is about to be perturbed to generate a new one in its neighbourhood, some of the characters in this solution are replaced with characters randomly selected from the rest of the input set, including $'\sharp'$.

5.2.2 Fitness definition

A key issue is to define a fitness function to (efficiently) evaluate the quality of solutions. This function should embody two aspects: (1) solutions should create as many discrete units as possible; (2) the solution should be as short as possible. The function needs to make a trade–off between these two points.

This work uses a function that rewards the early occurrence of discrete partitions and punishes the chromosome's length. An alternative would be to model the number of state partitions and the length of a solution as two objectives and then treat them as multi–objective optimisation problems (for more information on multi–objective optimisation problems with GA see, for example, ref. (Gol89)).

A fitness function is defined to evaluate the quality of an input sequence. While applying an input sequence to a finite state machine, at each stage of a single input, the state splitting tree constructed is evaluated by equation 5.1,

$$f_{(i)} = \frac{x_i e^{(\delta x_i)}}{l_i^{\gamma}} + \alpha \frac{(y_i + \delta y_i)}{l_i}. \tag{5.1}$$

where i refers to the i^{th} input character. x_i denotes the number of existing discrete partitions while δx_i is the number of new discrete partitions caused by the i^{th} input. y_i is the number of existing separated groups while δy_i is the number of new groups. l_i is the length of the input sequence up to the i^{th} element (***Do Not Care*** characters are excluded). α and γ are constants. It can be noted that a partition that finds a new discrete unit creates new separated groups as well.

Equation 5.1 consists of two parts: exponential part, $f_{e(i)} = \frac{x_i e^{(\delta x_i)}}{l_i^{\gamma}}$, and linear part, $f_{l(i)} = \alpha \frac{(y_i + \delta y_i)}{l_i}$. It can be seen that the occurrence of discrete partitions makes x_i and δx_i increase. Consequently, $x_i e^{(\delta x_i)}$ is increased exponentially.

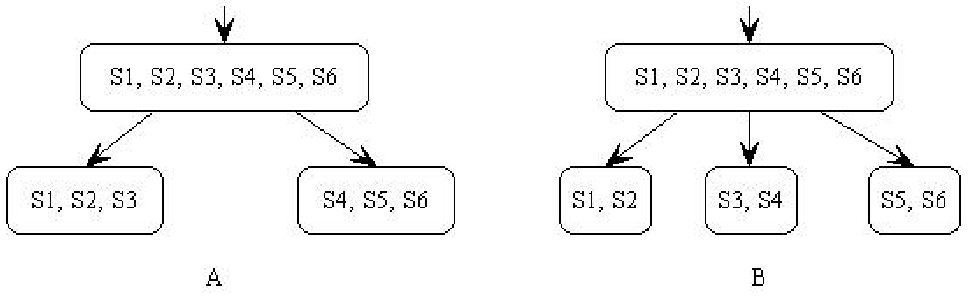


Figure 5.1: Two patterns of partitions.

Meanwhile, with the input sequence's length l_i increasing, l_i^γ is increased exponentially (γ should be greater than 1). Suppose x_i and l_i change approximately at the same rate, that is $\delta x_i \approx \delta l_i$, as long as $e^{(\delta x_i)}$ has faster dynamics than l_i^γ, $f_{e(i)}$ increases exponentially, causing f_i to be increased exponentially. However, if, with the length of the input sequence increasing, no discrete partition is found, $f_{e(i)}$ decreases exponentially, causing f_i to be decreased exponentially. $f_{e(i)}$ thus performs two actions: encouraging the early occurrence of discrete partitions and punishing the increment of an input sequence's length.

$f_{l(i)}$ also affects $f_{(i)}$ in a linear way. Compared to $f_{e(i)}$, it plays a less important role. This term rewards partitioning even when discrete classes have not been produced. Figure 5.1 shows two patterns with no discrete partition. We believe pattern **B** is better than **A** since **B** might find more discrete units in the forthcoming partitions.

Individuals that find discrete partitions at the first several inputs but fail to find more in the following steps may obtain higher fitness values than others. They are likely to dominate the population and cause the genetic pool to converge prematurely. To balance the evaluation, after all input characters have been examined, the final fitness value for an input candidate is defined as the average of equation 5.1

$$F = \frac{1}{N}\sum_{i=1}^{N} f_{(i)}. \tag{5.2}$$

where N is the sequence's length.

5.2.3 Application of sharing techniques

When constructing multiple UIOs using MOTs, solutions of all UIOs might form multi-modals (local optima) in the search space – a search might find only a few of these local optima and thus miss some UIOs. In order to find more UIOs, it is necessary to use some techniques to effectively detect local optima.

In this work, a sharing technique is applied. The fitness computation for candidates that are highly similar to others is guided for reduction. Degraded individuals are less likely to be selected for the reproduction. By using such an operation, the population in the genetic pool should be forced to form several sub-populations, each being used to identify a local optima.

A: Similarity Measurement

Before reducing a candidate's fitness, a mechanism should be used to evaluate the similarities between two solutions. There are two standard techniques that are proposed to measure the distance between two individuals, namely *Euclidian* distance and *Hamming* distance. However, both methods are not suitable in this work since inputs for a finite state machine are ordered sequences. The order of characters plays a very important role in evaluating the similarity. This work defines a Similarity Degree (SD) to guide the degradation of a candidate's fitness value.

Definition 5.2.1 *A valid partition (VP) is defined as a partition that gives rise to at least one new separated group when responding to an input character.*

Figure 5.2 illustrates two patterns of partition. In the figure, **A** is valid since the parent group is split into two new groups, while **B** is invalid since the current group is identical to its parent group and no new group is created. A UIO can be formed by a mixture of valid and invalid partitions.

Definition 5.2.2 *The maximum length of valid partition (MLVP) is the length up to an input that gives rise to the occurrence of the last valid partition.*

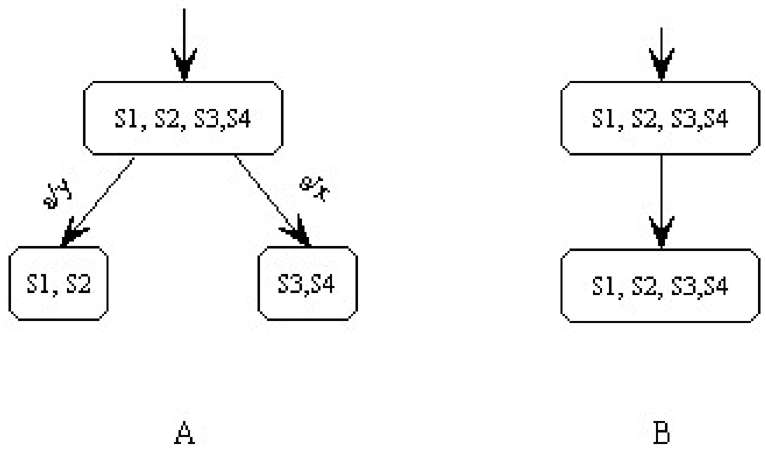


Figure 5.2: Patterns of valid partition (A) and invalid partition (B).

Definition 5.2.3 *The maximum discrete length (MDL) is the length up to an input character that gives rise to the occurrence of the last discrete partition.*

Since a discrete partition defines a valid partition, MDL can never be greater than MLVP in a state splitting tree.

Definition 5.2.4 *The Similarity Degree (SD) between two ordered sequences is defined as the length of a maximum length prefix sequence of these two sequences.*

If elements in two ordered sequences are the same before the N^{th} character and different at the N^{th}, the SD is $N - 1$ ($\sharp$ is excluded from the calculation). For example, the SD between $a\#b\#aaca\#\#a$ and $ab\#a\#bc\#a\#a$ is 3.

B: Fitness Degrade

In order to prevent the population from converging at one or several global optima in the search space, at each iteration of computation, some candidates (that are not marked as degraded) that have high *SD* value should have the fitness value reduced by the mechanism as follows:

1. if a candidate's *SD* is greater than or equal to its *MDL*, its fitness value should be degraded to a very small value; else

2. if $SD/MLVP$ passes a threshold value Θ, its fitness value is reduced to $(1 - SD/MLVP) * V_{Org}$, where V_{Org} is its original value.

If a candidate's SD is greater than or equal to its MDL, it implies that, in terms of finding discrete partitions, this solution has been significantly represented by others and becomes redundant. Generally, the fitness value of a redundant candidate needs to be zero to keep it from reproduction. However, in the experiments, we set the value to 1% of its original value, allowing it to be selected with a low probability. If not, $(1 - SD/MLVP) * V_{Org}$ controls the degree of decrement. The more information in a candidate that is represented in others, the more it is reduced. After a candidate's fitness value is reduced, it is marked as "Degraded".

Since a discrete partition defines a valid partition, MDL can never be greater than MLVP ($MDL \leq MLVP$). $(1 - SD/MLVP) * V_{Org}$ is applied only when $SD < MDL$. Since $SD < MDL \leq MLVP$, $(1 - SD/MLVP) * V_{Org}$ is positive. When SD is greater than or equal to MDL, a fitness is reduced to a small value but still positive. So, the fitness value of an individual is always positive.

Threshold value Θ might vary between different systems. Since it is enabled only when SD is less than MDL, a value between 0 and 1 can be applied. In model **III**, we used 2/3, while, in model **II**, we used 1/2.

5.2.4 Extending simple simulated annealing

A simple Simulated Annealing (SA) works on a single solution. In order to find all possible UIOs, multi-run based SA needs to be applied. Several researchers have studied the multi-run based SA (Atk92; MMSL96).

In this work, population based simulated annealing (PBSA) is used. Each individual in the genetic pool refers to a solution and is perturbed according to the simple SA scheme. All individuals in a population follow the same temperature drop control. During the computation, individuals in the genetic pool are compared with others. Those individuals that have been significantly represented by others have the fitness value reduced according to the sharing scheme.

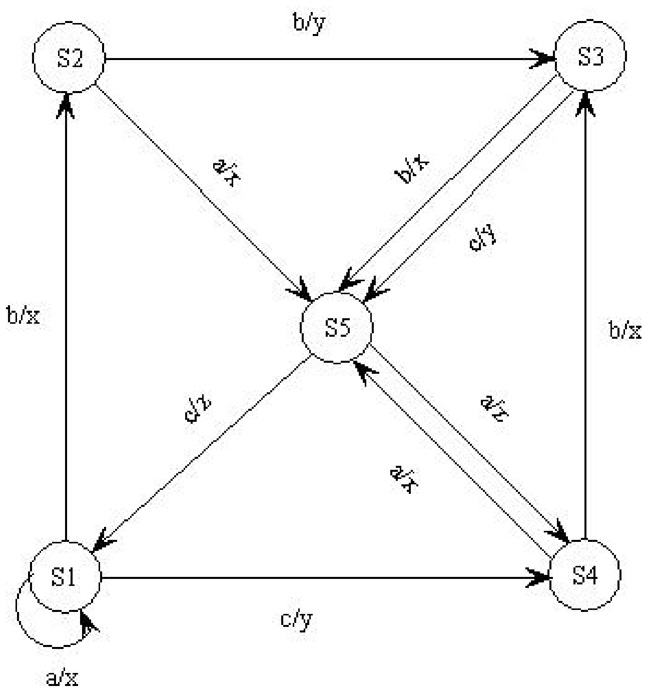


Figure 5.3: The first finite state machine used for experiments: model **I**.

5.3 Models for experiments

The study of the proposed model consists of two stages. In the first stage, the construction of UIOs using GAs was initially investigated. The performance of GAs was simply compared with that of random search. In the second stage, we investigated the impact of the sharing technique when constructing multiple UIOs using GAs and SA, and reported the experimental results.

Three models are used for experiments. These models are minimal and strongly connected. They are shown in Figure 5.3 (model **I**), Figure 5.4 (model **II**) and Figure 5.5 (model **III**), respectively.

The first model has 5 states. The input set is $I = \{a, b, c\}$ and the output set is $O = \{x, y\}$; the second model has 10 states and the third model has 9 states. Both the second and the third machines use the same input and output sets. They are: $I = \{a, b, c, d\}$ and $O = \{x, y, z\}$. The complete set of the minimum-length UIOs for model **I** is listed in Table 5.1.

When investigating the impact of the sharing techniques, in order to compare

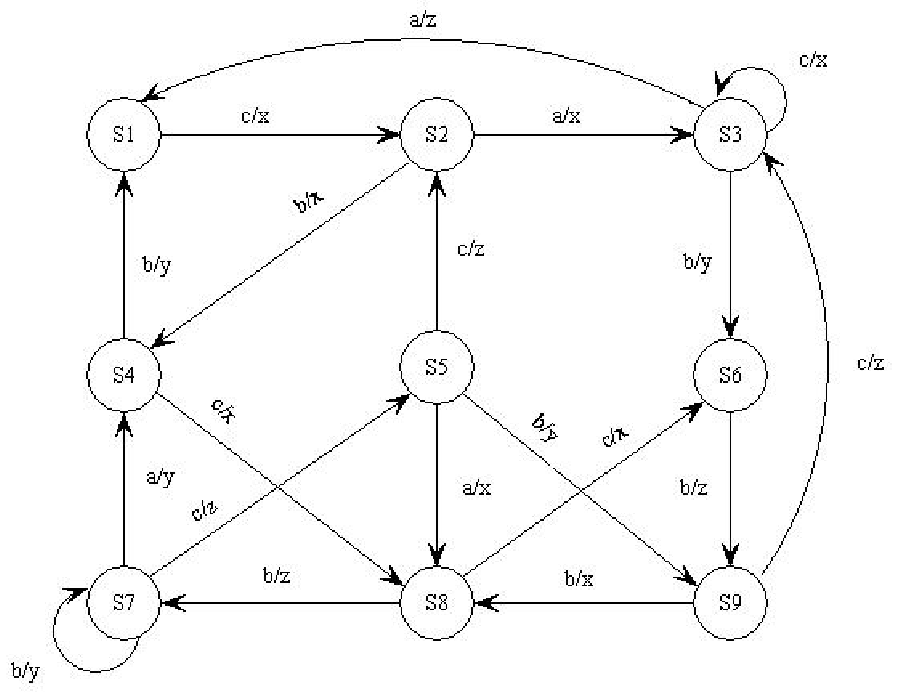


Figure 5.4: The second finite state machine used for experiments: model **II**.

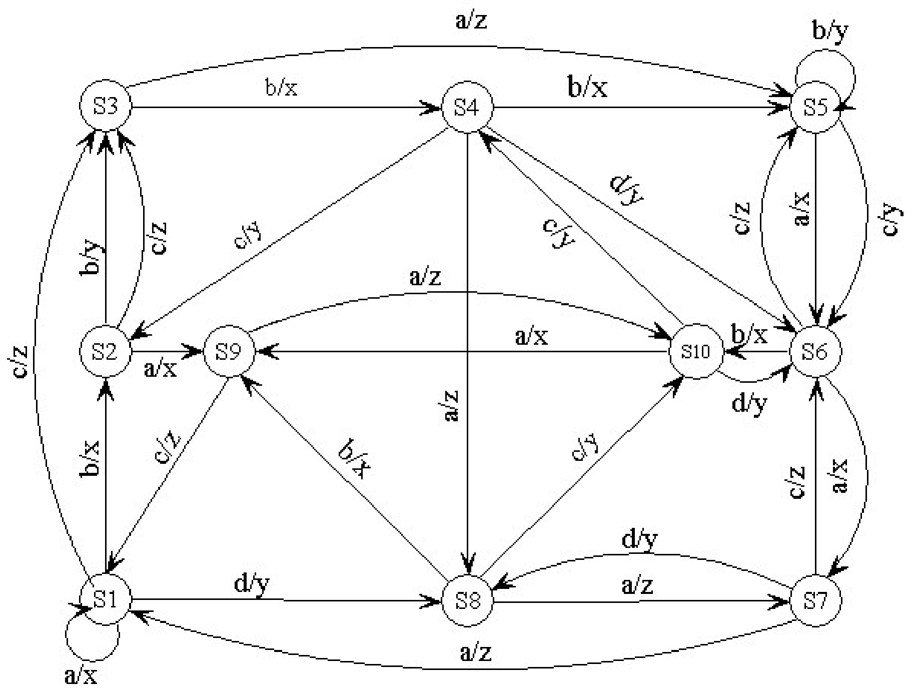


Figure 5.5: The third finite state machine used for experiments: model **III**.

State	UIOs
s_1	aa/xx, ab/xx, ac/xy, ba/xx bb/xy, ca/yx, cb/yx
s_2	b/y
s_3	ba/xz, bc/xz, ca/yz, cc/yz
s_4	bb/xx, bc/xy
s_5	a/z, c/z

Table 5.1: The minimum-length UIOs for model **I**.

SQ	NS	SQ	NS	SQ	NS	SQ	NS
cbb	7	cbca	6	bca	5	cacc	4
bcb	7	bcc	6	ccb	4	caca	4
bb	7	cacb	5	cbc	4	caa	4
cbcc	6	vab	5	cba	4	acb	4
ab	4	ca	3	aa	3	a	2
ccc	3	bcbc	3	cc	2	bc	1
cca	3	acc	3	bcba	2	-	-
cb	3	aca	3	ba	2	-	-

Table 5.2: UIOs for model **II**.

the set of UIOs produced with a known complete set, the search is restricted to UIOs of length 4 or less. With such a restriction, for model **II** and model **III**, $4^4 = 256$ input sequences can be constructed. There are 30 UIOs for model **II**, listed in Table 5.2 and 86 UIOs for model **III**, listed in Table 5.3. In the tables, SQ stands for the input from a UIO sequence and NS refers to the number of states this sequence can identify.

5.4 Working with genetic algorithms

5.4.1 GA vs. random search

In this section, a set of experiments was devised to investigate the performance of a simple GA on the construction of UIOs. The experiments were designed by using model **I** first, and then model **II**. The performance of a simple GA was experimentally compared with that of random search.

SQ	NS	SQ	NS	SQ	NS	SQ	NS
aaaa	8	aaa	5	bcca	5	accd	4
aaab	7	aabc	5	bccb	5	bab	4
cca	7	aacc	5	cbcd	5	baca	4
ccb	7	abca	5	ccc	5	bacb	4
aaad	6	acaa	5	aabb	4	bbc	4
aaca	6	acad	5	aabd	4	bca	4
acca	6	baa	5	abcc	4	bcba	4
accb	6	bba	5	abcd	4	bcbc	4
accc	6	bbb	5	aca	4	bcbd	4
cbca	6	bcad	5	acba	4	bccc	4
cbcc	6	bcbb	5	acbb	4	caaa	4
caab	4	bcc	3	aadc	2	ca	1
cbaa	4	bcd	3	abd	2	aad	1
cbaa	4	bcd	3	abd	2	aad	1
cbb	4	caac	3	acbd	2	acb	1
cbcb	4	cabc	3	ba	2	ada	1
aab	3	cacc	3	caa	2	adba	1
aba	3	cbab	3	cabb	2	adbc	1
abb	3	cbd	3	cabd	2	adbd	1
abc	3	aacb	2	caca	2	adc	1
bacc	3	ccb	2	cacb	2	cd	1
bad	3	aacd	2	cad	2	bb	3
aada	2	cb	2	-	-	-	-

Table 5.3: UIOs for model **III**.

A: Tracking historical records

When implementing GAs, mutation prevents the computation from getting stuck in a local maxima/minima but might also force it to jump out of the global maxima/minima when it happens to be there. Solutions provided at the end of evolutionary computation could be good, but need not be the best found during the process. It is therefore useful to keep track of those candidates that have produced good or partially good solutions, and store them for the purpose to further optimise the final solutions.

Consider an example shown in Figure 5.6. Suppose that a GA produces a UIO sequence U_t for state s_t, forming a path shown in thin solid arrow lines. During the computation, another solution U'_t for s_t has been found, forming a path shown in dotted arrows. The two lines visit a common node at N_4. U_t has a shorter path than U'_t before N_4 while has a longer path after N_4. The solution recombined from U_t and U'_t (indicated in figure by thick arrow lines), taking their shorter parts, is better than either of them.

In the study of computing UIOs using a simple GA, a database is used to track candidates that result in the occurrence of discrete partitions. This database is then used to further optimise the final solutions through recombination. Solutions for a state, which are of the same length, are multi-UIOs for this state which can be used in test generation proposed in refs. (SLD92; YU90).

B: Experiments with model I

Goldberg (Gol89) has studied the effects of different choices of crossover and mutation rates. In this work, we simply follow his suggestions. The parameters are set to: [1]$ChrLen = 10$, $XRate = 0.75$, $MRate = 0.05$, $PSize = 30$, $MGen = 50$, $\alpha = 15$, and $\gamma = 1.5$. The settings of crossover rate and mutation rate remain unchanged throughout all GA experiments.

Model **I** is first used for the experiments. All minimum-length UIOs for all states are presented in Table 5.1. Roulette Wheel Selection (RWS) and Uniform Crossover (UC) are implemented.

[1]ChrLen:Chromosome Length; XRate:Crossover Rate; MRate:Mutate Rate; PSize: Population Size; MGen:The Maximum Number of Generations.

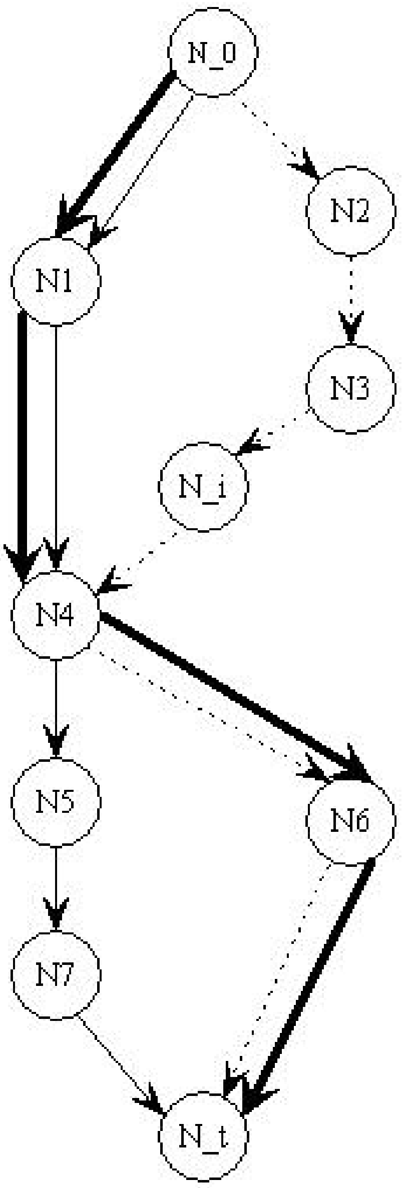


Figure 5.6: Solution recombination.

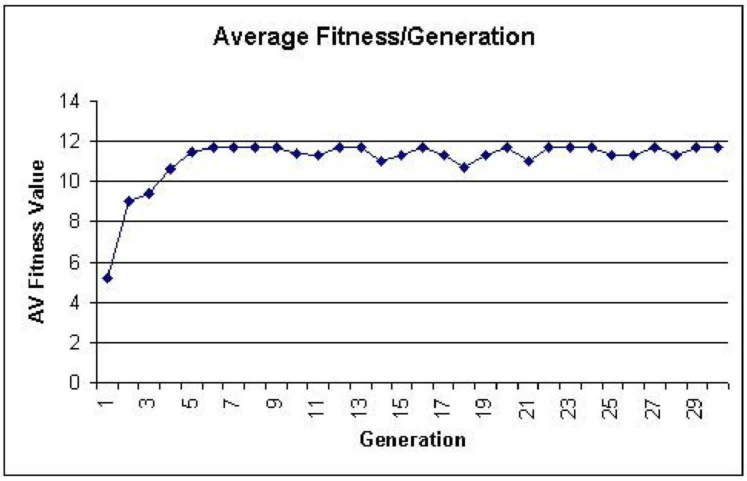


Figure 5.7: Average fitness value - input space {a,b,c}.

At the end of computation, by looking at the average fitness values (Figure 5.7), we found that the genetic pool converges quite quickly. The curve is comparatively smooth. However, by examining all individuals (Table 5.4), we found that the whole population tends to move to the individuals that start with *bb* or *bc*. The population loses its diversity and converges prematurely. Consequently, only a few UIOs have been found $\{b, bb, bc\}$.

This is not what we expected. To keep the genetic pool diverse, we introduced a ***DO NOT CARE*** character '$\sharp$'. When receiving this character, the state of an FSM remains unchanged. The input space is then $\{a, b, c, \sharp\}$. We keep the same values for all other parameters. The average fitness chart is presented in Figure 5.8. It can be seen that Figure 5.8 is not as smooth as Figure 5.7, but still shows a general tendency to increase. After examining the genetic pool, we found that eleven UIOs, $\{a, b, c, ab, ac, ba, bb, bc, ca, cb, cc\}$, were found ([1]Table 5.5).

By retrieving the historical records, we also found $\{aa\}$ (Table 5.6). The GA thus performs well in this experiment.

The reason that crossover can be used to explore information is that it forces genes to move among chromosomes. Through recombination of genes, unknown

[1]*Sequence*: candidate sequence. VS: minimum-length UIO.

ID	Sequence	ID	Sequence
1	*bccaabcacc*	16	*bbcaabcacc*
2	*bbbaabaacc*	17	*bbbaabaaca*
3	*bbbaabcacc*	18	*bbcaababcc*
4	*bcccabaacb*	19	*bbbcabbacc*
5	*bbcbabcacb*	20	*bbcaabaacc*
6	*bbbaabcacc*	21	*bccaabaacc*
7	*bbcbabcacb*	22	*bbbaabaacc*
8	*bbccabcacb*	23	*bbbcabbacc*
9	*bbcbabaacb*	24	*bbcbabcacb*
10	*bcbbabaacb*	25	*bbbcabcacc*
11	*bbcbabcabb*	26	*bbcaabcacb*
12	*bbcaabaacc*	27	*bbbbababcc*
13	*bccaabaacb*	28	*bbcaabcacb*
14	*bbbbabcacc*	29	*bbccabbacc*
15	*bbcbabcacb*	30	*bbbaabbacb*

Table 5.4: Final sequences obtained from model **I** - input space {a,b,c}.

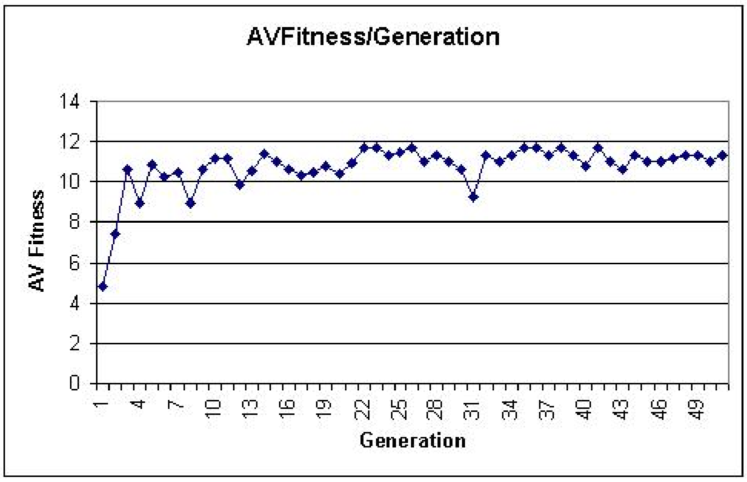


Figure 5.8: Average fitness value - input space {a,b,c,♯}.

ID	Sequence	VS	ID	Sequence	VS
1	$bbbbcccb\sharp c$	bb	16	$\sharp bcbabcabb$	bc
2	$ab\sharp bbc\sharp bbc$	ab	17	$\sharp\sharp cba\sharp cbcb$	cb
3	$\sharp bc\sharp caaaba$	bc	18	$\sharp bb\sharp\sharp bbaca$	bb
4	$\sharp\sharp bcacbc\sharp c$	bc	19	$bc\sharp bb\sharp cbb\sharp$	bc
5	$b\sharp bcaca\sharp cb$	bb	20	$cbcbbc\sharp\sharp ca$	cb
6	$bb\sharp bca\sharp\sharp bb$	bb	21	$\sharp bc\sharp aabc\sharp c$	bc
7	$cbacc\sharp ac\sharp b$	cb	22	$c\sharp a\sharp cacc\sharp c$	ca
8	$acabaaaacc$	ac	23	$c\sharp accac\sharp\sharp a$	ca
9	$ba\sharp a\sharp aaa\sharp b$	ba	24	$\sharp b\sharp bcabbc\sharp$	bb
10	$\sharp ccaab\sharp acc$	cc	25	$bb\sharp bba\sharp\sharp a\sharp$	bb
11	$\sharp cbc\sharp aa\sharp ab$	cb	26	$bacb\sharp c\sharp b\sharp b$	ba
12	$\sharp bbcca\sharp aaa$	bb	27	$bb\sharp bb\sharp\sharp\sharp c\sharp$	bb
13	$cccb\sharp\sharp\sharp\sharp\sharp c$	cc	28	$cb\sharp babc\sharp b\sharp$	cb
14	$aca\sharp bbaa\sharp b$	ac	29	$cccc\sharp cc\sharp bb$	cc
15	$cbb\sharp c\sharp cb\sharp c$	cb	30	$b\sharp cca\sharp\sharp b\sharp c$	bc

Table 5.5: Final sequences obtained from model **I** - input space {a,b,c,♯}.

ID	Sequence	VS	Fitness
1	$aa\sharp caab\sharp c$	aa	6.2784
2	$a\sharp aba\sharp bc\sharp c$	aa	4.2605

Table 5.6: Solutions obtained from the historical record database.

Exp.	UIOs Found	Total	Percent(%)
1	8	12	66.7
2	8	12	66.7
3	11	12	91.7
4	11	12	91.7
5	10	12	83.3
6	11	12	91.7
7	11	12	91.7
8	11	12	91.7
9	10	12	83.3
10	12	12	100
11	11	12	91.7
Avg	10.36	12	86.4

Table 5.7: Average result from 11 experiments.

information can be uncovered by new chromosomes. However, the gene movement exerted by crossover can only happen among different chromosomes. We call it vertical movement. By using a *DO NOT CARE* character, some spaces can be added in a chromosome, which makes it possible for genes to move horizontally. Therefore, *DO NOT CARE* makes the exploration more flexible, and, consequently, can help to keep the genetic pool diverse.

We organised eleven experiments with the same parameters. By examining the solutions obtained in the final genetic pool (historical records are excluded), we evaluated the average performance. Table 5.7 shows that, in the worst case, 8 out of 12 UIOs are found, which accounts for 66.7%. The best case is 100%. The average is 86.4%.

After examining the solutions from different experiments, we found that *aa* is the hardest UIO to be found while *bb* and *bc* are most frequent ones that occur in the final solutions. By checking Table 5.1, we found a very interesting fact: a majority of UIOs initially start with *b* or *c*. If individuals happen to be initialised with $ba \times \times \times \times \times \times \times \times$, they will distinguish s_1, s_2 and s_5 in the first two steps, and so achieve high fitness. These individuals are likely to be selected for the next generation. Individuals initialised with $aa \times \times \times \times \times \times \times \times$ can distinguish only s_1 and s_5 in the first two steps, and achieve lower fitness values. They are

ID	Sequence	VS	ID	Sequence	VS
1	*cabbbcbcac*	*ca*	16	*cacacacccc*	*ca*
2	*ccbcbbbcbc*	*cc*	17	*bcbbabcbca*	*bc*
3	*bcccaaabcb*	*bc*	18	*ccccbbcccb*	*cc*
4	*cccccbcbac*	*cc*	19	*accbbccacb*	*ac*
5	*ccbbacbccc*	*cc*	20	*cbccaccccb*	*cb*
6	*bccccabcac*	*bc*	21	*acbbcbbbbb*	*ac*
7	*bccbcbbcbc*	*bc*	22	*cbccccbccb*	*cb*
8	*ccbbccaabc*	*cc*	23	*ccabacbcca*	*cc*
9	*bbaccccba*	*bb*	24	*accacbabcc*	*ac*
10	*cbbbbacccb*	*cb*	25	*aabcbacbbb*	*aa*
11	*bcaaccccb*	*bc*	26	*cabbacacbc*	*ca*
12	*cccbcbbcbc*	*cc*	27	*bcbccccaac*	*bc*
13	*abcbbcccab*	*ab*	28	*aabcbccbca*	*aa*
14	*bcccccbccc*	*bc*	29	*bccccaabbb*	*bc*
15	*bbccbccbba*	*bb*	30	*acaccbaaba*	*ac*

Table 5.8: Solutions obtained using by random search.

less likely to be selected for reproduction. This fact seems to imply that there exist multiple modals in the search space. Most individuals are likely to crowd on the highest peak. Only a very few individuals switch to the lower modals. To overcome this problem, *sharing techniques* might help. The application of such approaches is studied in the following subsections.

We then turn to compare the performance between GA and random search. Random search is defined as randomly perturbing one bit in an input sequence. 30 input sequences of ten input characters were randomly generated. We repeated this experiment 10 times. The results shown in Table 5.8 are the best ones. From the table it can be seen that 11 out of 12 UIOs (a, b, c, aa, ab, ac, bb, bc, ca, cb, cc) are found over these experiments. Only one is missed (*ba*).

B: Experiments with model II

Since model **I** is comparatively simple, and the UIOs are short, it is not difficult to find all UIOs through random search. Thus, the GA does not show significant advantages over random search. A more complicated system, model **II**, is therefore used to further test GA's performance.

State	UIOs
s_1	ca/xx, cb/xx
s_2	aa/xz, ab/xy, acc/xxx, bb/xy bcc/xxx, bcba/xxzy
s_3	a/z, bb/yz, ca/xz, cb/xy, ccb/xxy, ccc/xxx
s_4	bc/yx, cba/xzy, ccb/xxz cbca/xzzx, cbcc/xzzz
s_5	ab/xz, acb/xxz, bb/yx bcc/yzx, cacc/zxxx, cb/zx
s_6	bb/zx, bcc/zzx
s_7	a/y, bb/yy, bcc/yzz, bcbc/yzyz, cbc/zyz, cc/zz
s_8	bb/zy, bcc/zzz, bcbc/zzyz, cbca/xzzz, cbcc/xzzx
s_9	bb/xz, ca/zz, cc/zx

Table 5.9: UIO sequences for model **II** found by random search.

However, no existing UIOs are available for model **II**, which means that we can never be sure that a complete set of UIOs has been found. Hence, we will compare the numbers of UIOs found by using random search and GA separately.

A total of 50 candidates were used in the experiment. All UIOs found, whether minimum-length or not, are listed to make a comparison. Experiments on both random search and GA were repeated 10 times. The solutions presented in Table 5.9 and Table 5.10 are the best. Table 5.9 lists the UIOs obtained through random search while Table 5.10 shows the solutions found by GA. After comparing these two tables, we found that GA finds many more UIOs than random search does. Both random search and GA easily find the short UIOs. However, for other UIOs the performance of GA appears to be much better than that of random search. For example, GA finds *bcbcc/xxzzz* and *cbcbb/xzzyz* while random search does not.

We also measured the frequency of hitting UIOs. Random search is redefined by initialising population routinely. Experimental result show that both methods hit UIOs with the length of 3 or less frequently. However, on hitting those with the length of 4, random search is roughly the half times of GA, while, for those

State	UIOs
s_1	ca/xx, cb/xx
s_2	aa/xz, ab/xy, aca/xxz,acb/xxy acc/xxx, bb/xy, bcc/xxx, bcbcc/xxzzz
s_3	a/z, bb/yz, ca/xz, cb/xy, ccb/xxy, ccc/xxx
s_4	bc/yx, cba/xzy,cbcbb/xzzyx cbb/xzy, ccb/xxz, cbca/xzzx, cbcc/xzzz, cbcbc/xzzyz
s_5	ab/xz, acb/xxz, bb/yx, bca/yzz bcc/yzx, cab/zxy, cb/zx
s_6	bb/zx, bca/zzz, bcc/zzx
s_7	a/y, ba/yy, bb/yy, bca/yzx bcc/yzz, cab/zxz, cbb/zyx, cc/zz cbc/zyz, cc/zz, bcbc/yzyz
s_8	ba/zy, bb/zy, bca/zzx, bcc/zzz, cbb/xzx, cbca/xzzz, cbcc/xzzx, cbcbb/xzzyz
s_9	bb/xz, ca/zz, cbb/zyz, cc/zx

Table 5.10: UIO sequences for model **II** found by GA.

with the length of 5, in the first 30 iterations, random search hits 10 times while GA 27.

All these results suggest that, in simple systems, it is possible to obtain good solutions through random search. However, in more complicated systems, especially in those with large input and state spaces, finding UIOs with random search is likely to be infeasible. By contrast, GA seems to be more flexible.

C: summary

In this subsection, the performance on computing UIOs using simple GAs was investigated. It is shown that the fitness function can guide the candidates to explore potential UIOs by encouraging the early occurrence of discrete partitions while punishing length.

It is also demonstrated that using a *DO NOT CARE* character can help to improve the diversity in GAs. Consequently, more UIOs can be explored. The simulation results in a small system showed that, in the worst case, 67% of the minimum-length UIOs have been found while, in the best case, 100%. On the average, more than 85% minimum-length UIOs were found from the model under the test. In a more complicated system, GA found many more UIOs than random search. GA was much better than random search at finding the longer UIOs. These experiments and figures suggest that GAs can provide good solutions on computing UIOs.

However, it was also noted that some UIOs were missed with high probability. This may be caused by their lower probability distribution in the search space. The problem was further studied in the next subsection.

5.4.2 Sharing vs. no sharing

Experiments in this subsection investigate the impact of the sharing techniques that aim to overcome the problem discussed in section 5.4.1. Maximum generation is set to $MGen = 300$. The population size is set to $PSize = 600$. The value of $MGen$ and $PSize$ remain unchanged throughout all following experiments.

The experiments used model **III** first, and then model **II**. Threshold value θ is set to 2/3 for model **III** and 1/2 for model **II**. α and γ are set to 20 and 1.2 respectively (section 5.7 explains the reason for choosing such values).

The first experiment studied the UIO distribution when using GA without sharing. The experiment was repeated 10 times. Figure 5.9 shows the UIO distribution of the best result. It can be seen that a majority of individuals move to a sub–population that can identify 7 states. The rest scatter among some other sub–populations that can identify 8, 6, 5, 4, 3, 2, 1 states. Due to such an uneven distribution, some sequences that define UIOs of 6, 5, 4, 3, 2, 1 states are likely to be missed. Only 59 are found and so 27 were missed.

An experiment was then designed to investigate the use of the sharing technique. This experiment was repeated 10 times. The best result is shown in Figure 5.10 (in B–H, sequences of legends indicate input sequences that define UIOs). It can be seen that, after applying sharing, the population is generally spread out, forming 9 sub–populations. Each sub–population contains UIO sequences that identify 0, 1, 2, 3, 4, 5, 6, 7, 8 states correspondingly. Only 4 UIOs were missed – the performance of the search had improved dramatically. However, the distributions in sub–populations do not form a good shape. Each sub–population is dominated by one or several UIOs.

The impact of sharing techniques was further investigated by using model **II**. Figure 5.11 shows the best result from the 5 experiments. It can be seen that the distribution of input sequences is similar to that of GA with sharing in model **III**. Generally, the population is spread out, forming several sub–populations. However, each sub–population is dominated by several individuals. We found that 2 UIOs were missed.

The experimental results above suggest that, when constructing UIOs using a GA, without the sharing technique, the population is likely to converge at several individuals that have high fitness values. The distribution of such a population causes some UIOs to be missed with high probability. This is consistent with the results of section 5.4.1. After applying the sharing technique, the population is encouraged to spread out and forms several sub–populations. These sub–populations are intended to cover all optima in the search space. The search quality was significantly improved and more UIOs were found.

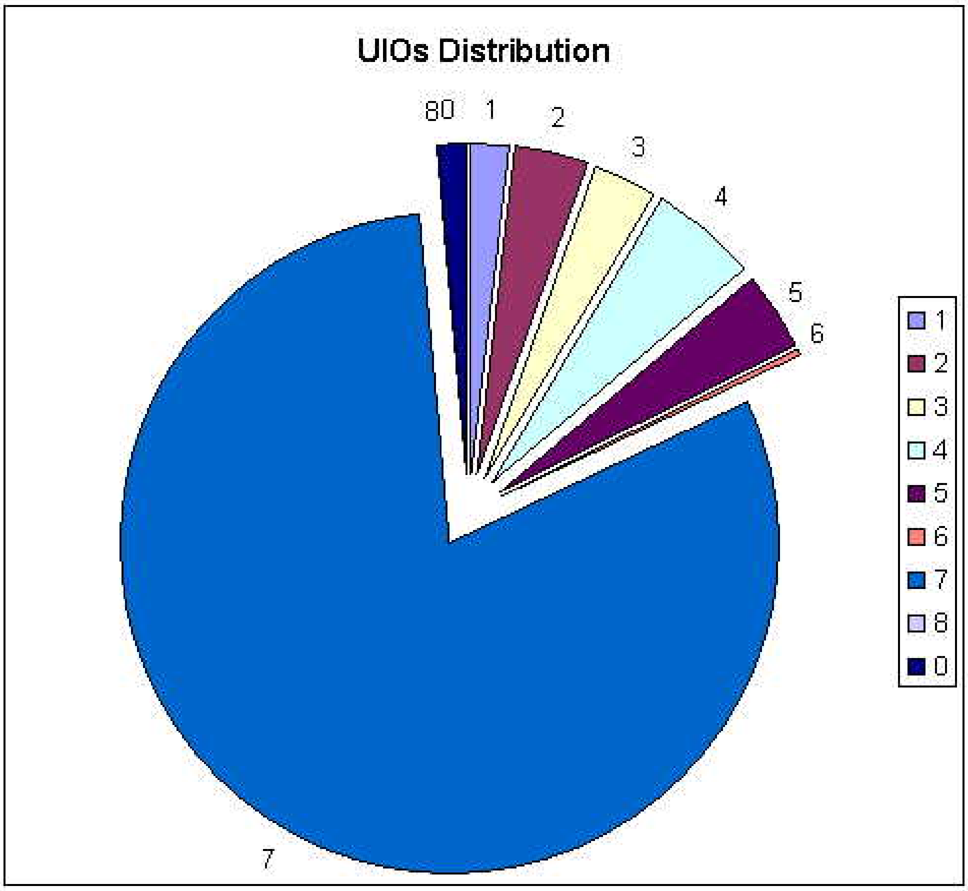


Figure 5.9: UIO distribution using GA without sharing for model **III**; Legends indicate the number of states that input sequences identify.

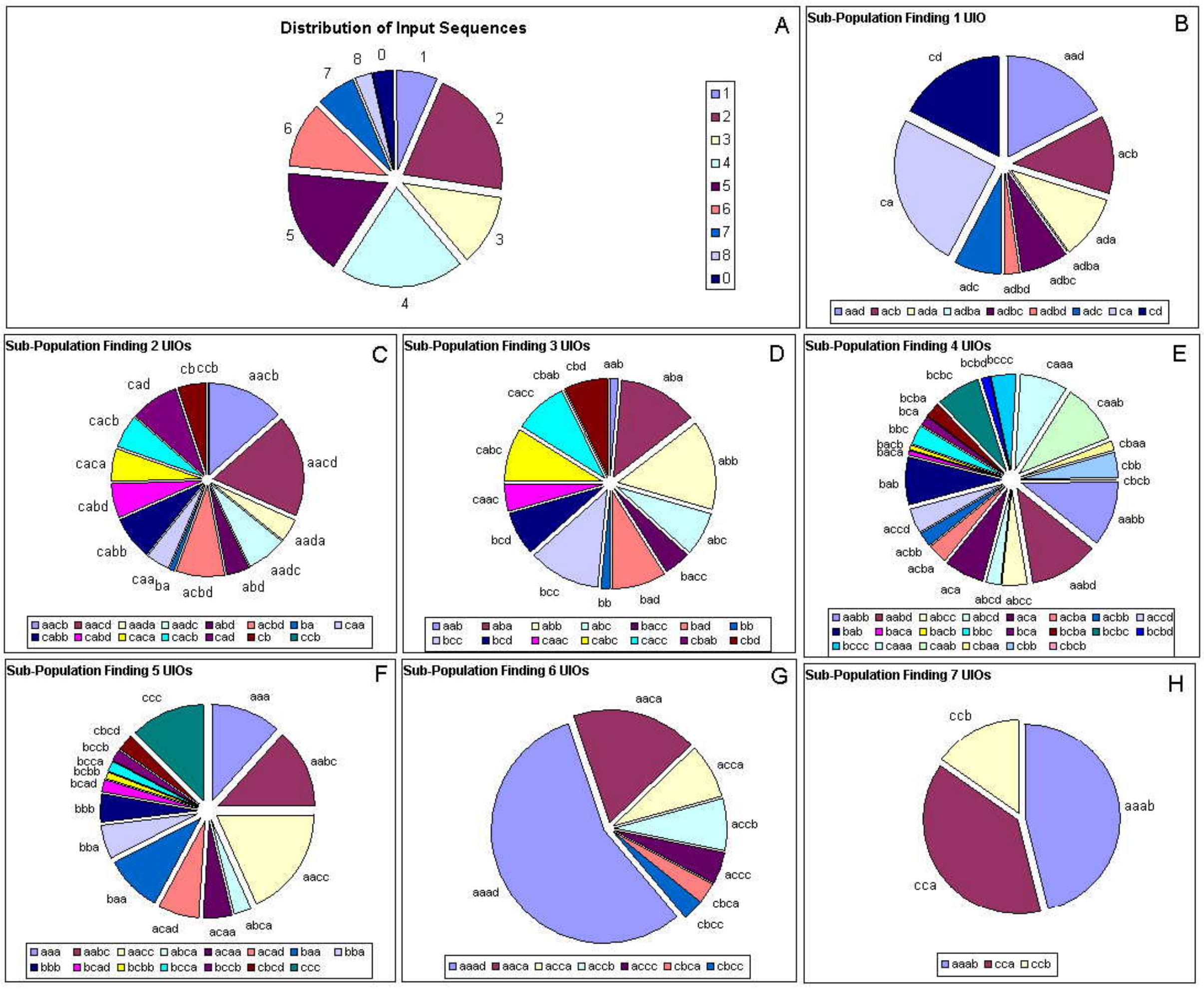


Figure 5.10: UIO distribution using GA with sharing for model **III**.

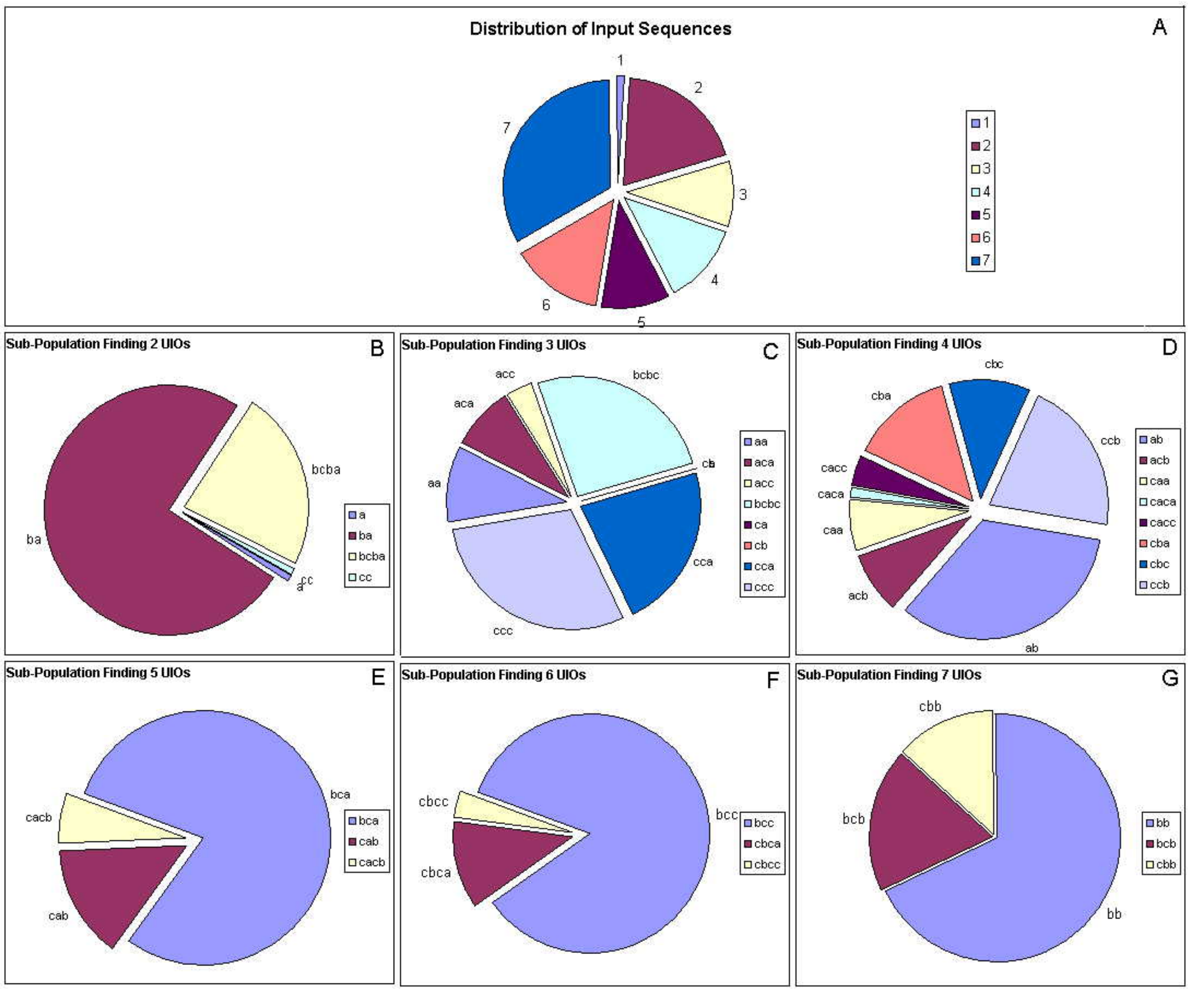


Figure 5.11: UIO distribution using GA with sharing for model **II**.

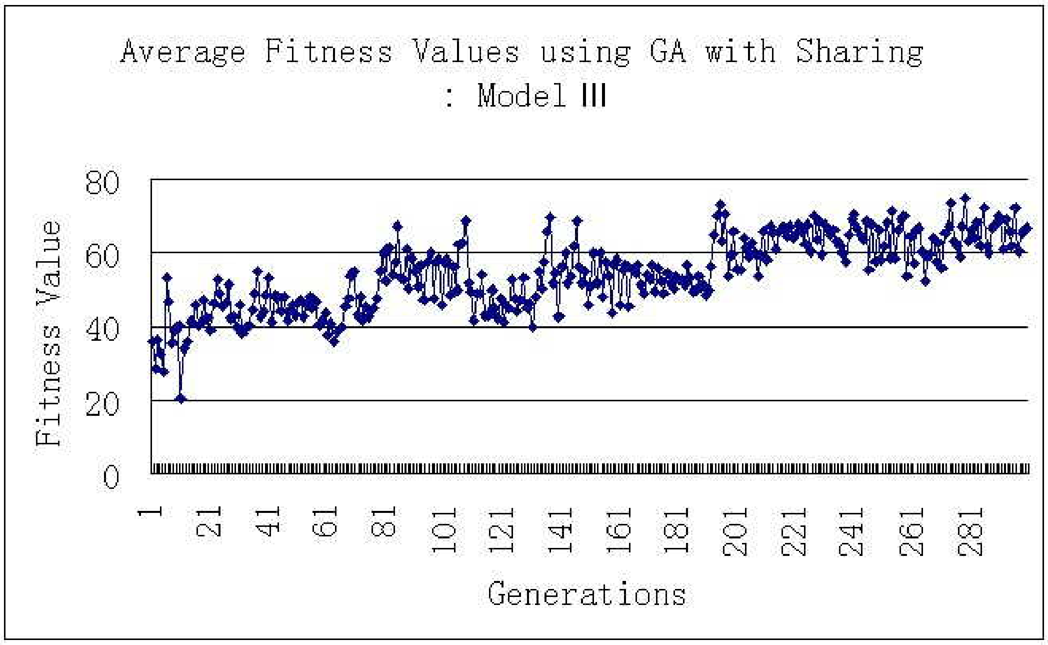


Figure 5.12: Average fitness values when constructing UIOs using GA for model **III**.

Convergence rates have also been studied when constructing UIOs for both models. Figure 5.12 and Figure 5.13 show the average fitness values when constructing UIOs for model **III** and model **II** respectively. From figures it can be seen that, in model **III**, the genetic pool begins to converge after 200 generations while, in model **II**, genetic pool converges after 60 generations.

5.5 Working with simulated annealing

Experiments described in this section aim to study the performance of SA. As described in section 5.2.4, a population based SA (PBSA) was used. Each individual in the genetic pool referred to a solution and was updated according to a simple SA's scheme. Individuals in the genetic pool were compared with others according to the sharing scheme. All individuals in a population followed the same temperature drop control. We also made a further restriction on the creation of a new solution. When an individual was required to generate a new solution, it was continuously perturbed in its neighbourhood until the new solu-

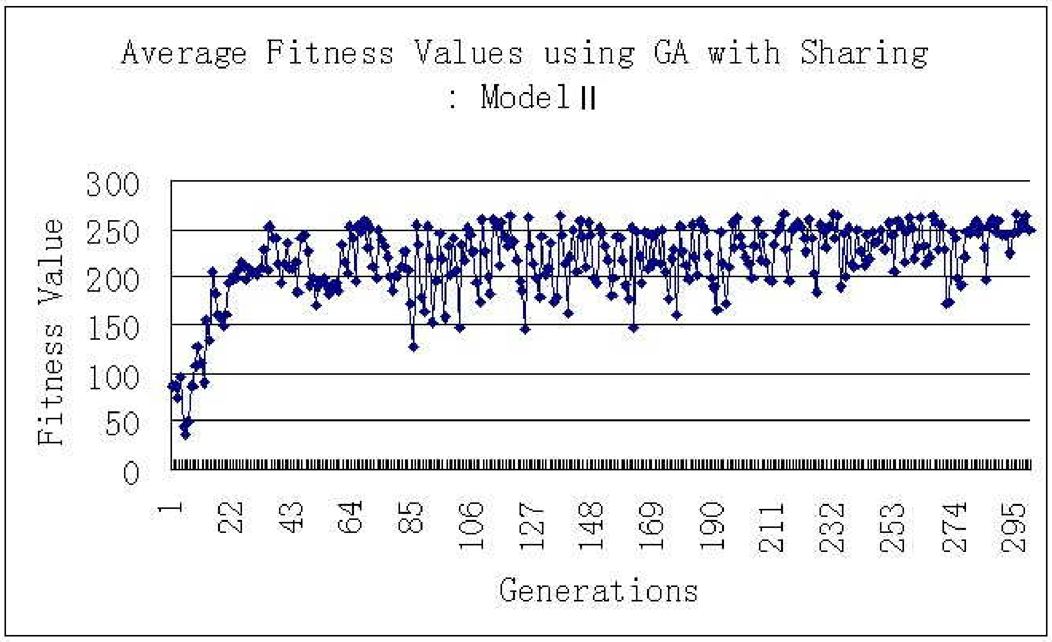


Figure 5.13: Average fitness values when constructing UIOs using GA for model **II**.

tion found at least one discrete partition. In order to make a comparison with GA, the maximum number of generations is always set to 300.

Model **III** was first applied for the experiments. Two temperature drop control schema were considered. In the first experiment, the temperature was reduced by a normal exponential function $nT(i+1) = 0.99 * nT(i)$ (Figure 5.14-A), and a sharing technique was applied. The experiment was repeated 10 times and the best result is shown in Figure 5.15.

From the figure it can be seen that the general distribution and sub–population distributions are quite similar to that of GA with sharing. The population was formed with several sub–populations. Each sub–population was dominated by several individuals. A total of 8 UIOs were missed. Compared to the experiments studied in section 5.4, this figure is quite high. In order to improve the search quality, the temperature drop scheme was changed to $nT(i+1) = 0.99 * nT(i) + nS(i+1) * sin(10 * \pi * i)$, where $nS(i+1) = 0.95 * nS(i)$. The curve of the function is shown in Figure 5.14-B. Generally, the tendency of temperature control is still exponentially decreasing, but local bumps occur. The best result from 10

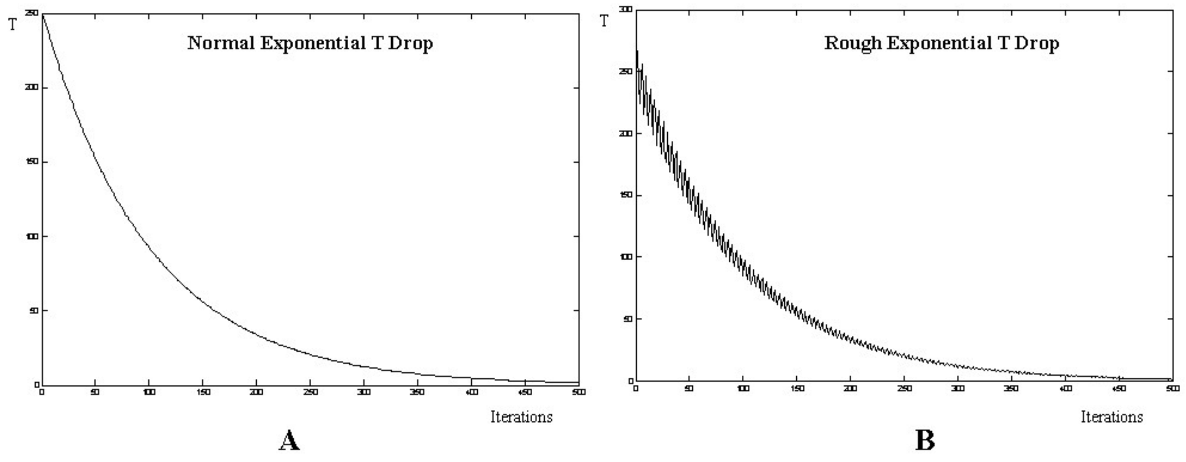


Figure 5.14: Simulated annealing temperature drop schema; A: normal exponential temperature drop; B: rough exponential temperature drop.

experiments is shown in Figure 5.16. We find that the distribution of population and sub–population have no significant changes. However, only 2 UIOs were missed. The performance is much better than the previous one.

The two SA methods were further studied by using model **II**. Figure 5.17 shows the best result using the normal temperature drop control while Figure 5.18 shows the best result for the rough temperature drop control. From these figures it can be seen that, compared to the experiments using model **III**, the distribution of input sequences have no significant changes. Both temperature control schemes achieve a good performance. In normal temperature drop experiment, 3 UIOs are missed while 2 UIO are missed in rough temperature drop experiment.

Figure 5.19 and Figure 5.20 present the average fitness values when constructing UIOs for model **III** and model **II** using rough temperature drop control scheme respectively. The figures show that, when using model **III**, the genetic pool begins to converge after 250 generations while, when using model **II**, the genetic pool converges after 200 generations. Comparing to Figure 5.12 and Figure 5.13, it can be seen that SA converges slower than GA.

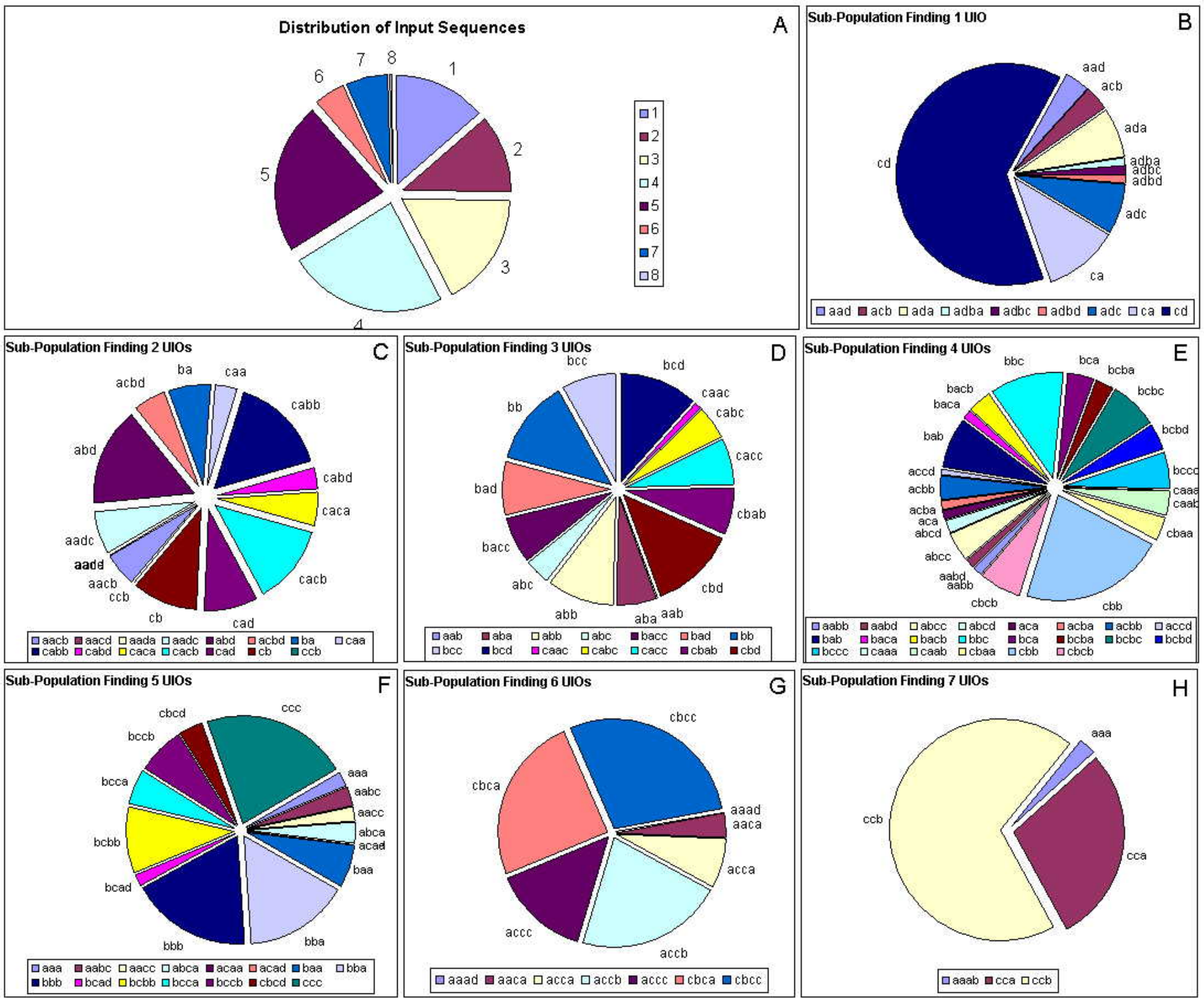


Figure 5.15: UIO distribution using SA with normal temperature drop for model **III**.

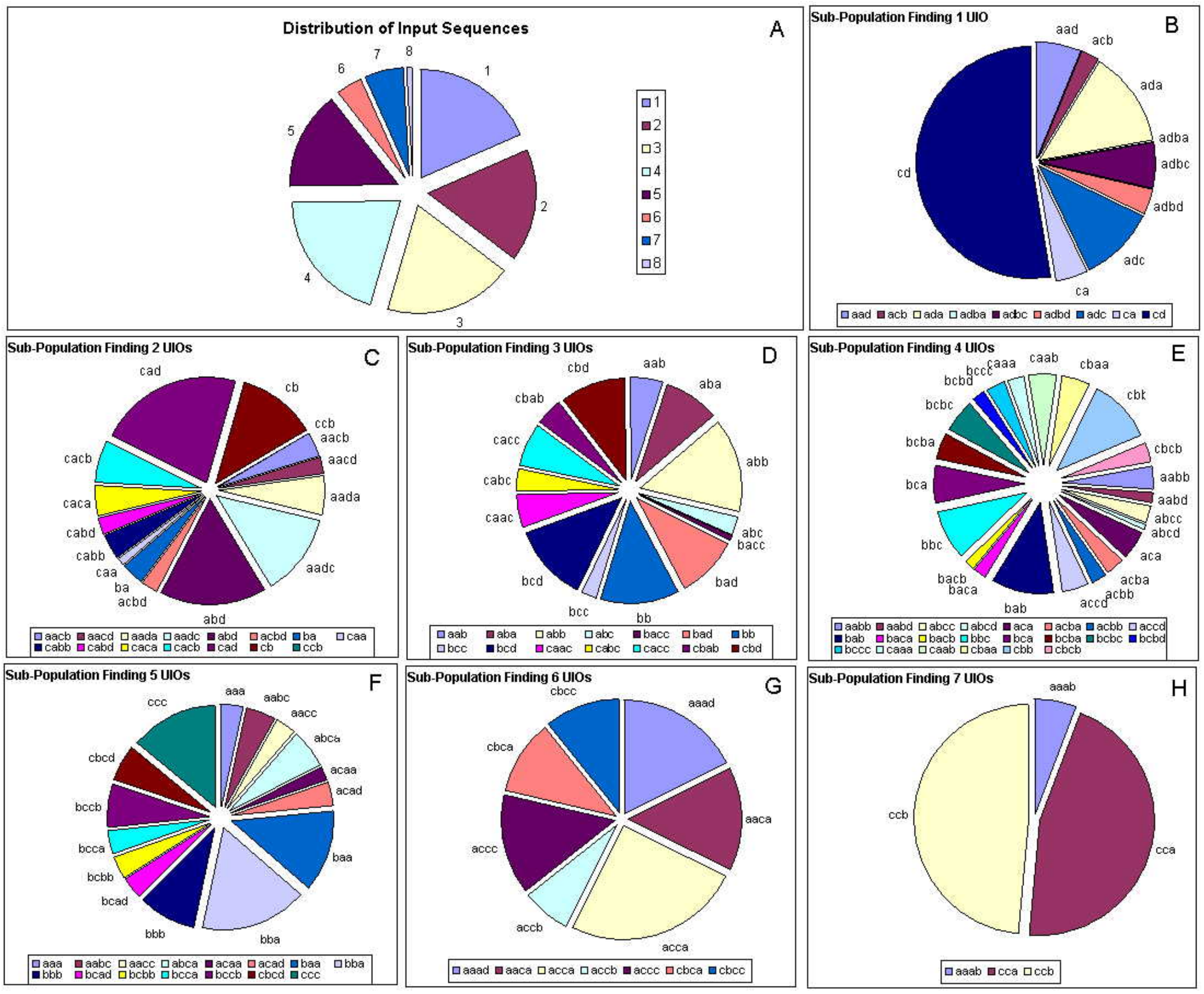


Figure 5.16: UIO distribution using SA with rough temperature drop for model **III**.

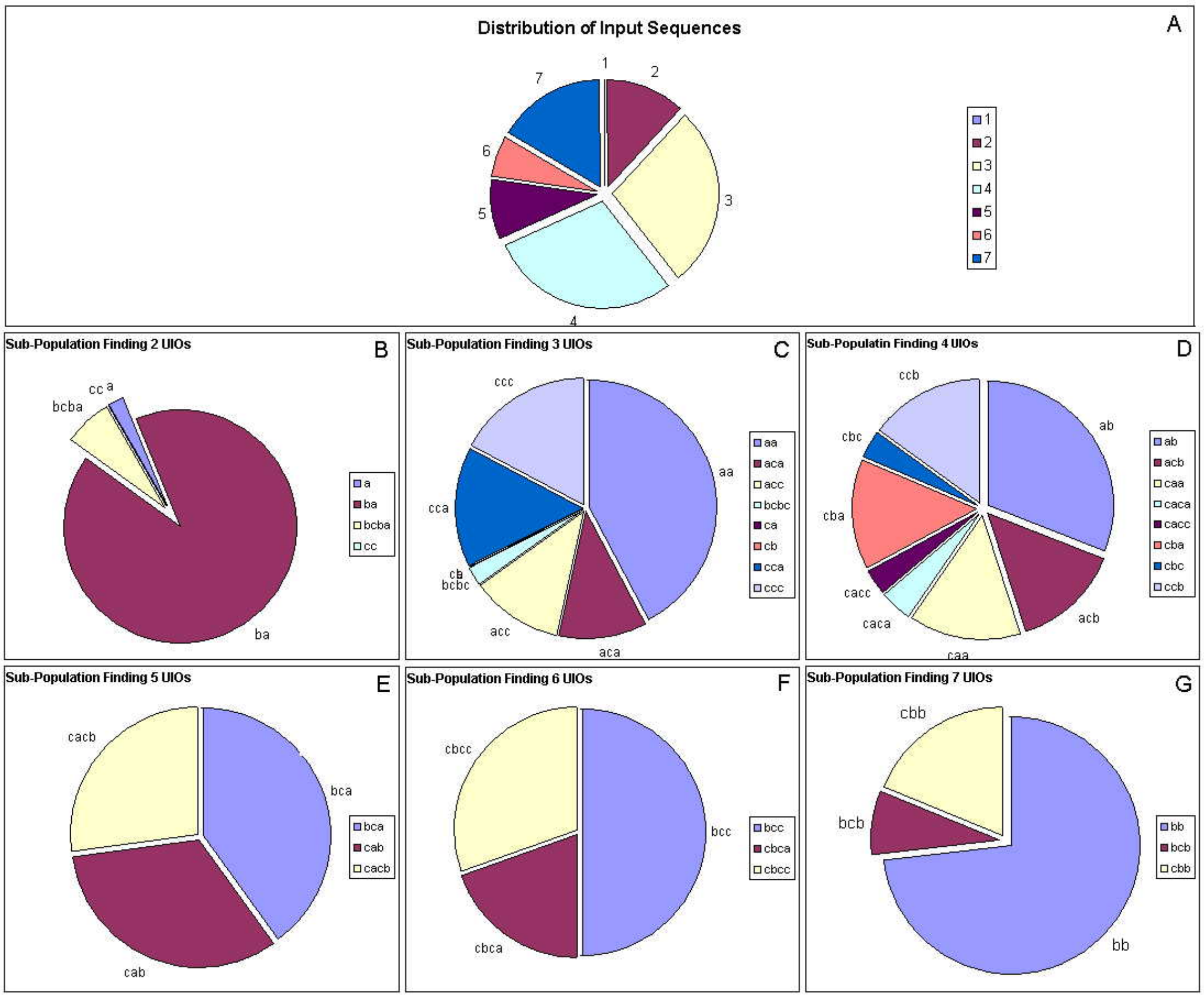


Figure 5.17: UIO distribution using SA with exponential temperature drop for model **II**.

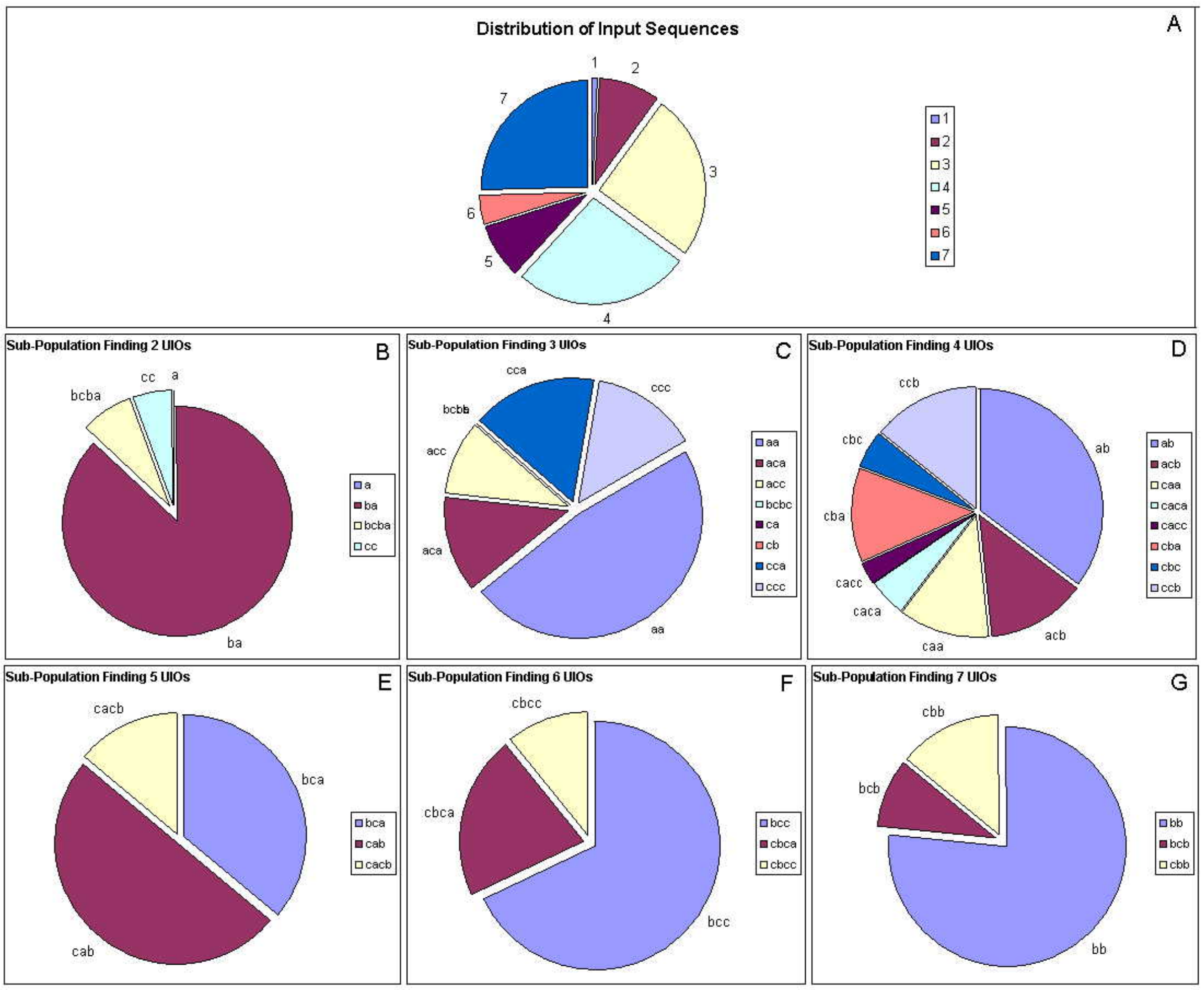


Figure 5.18: UIO distribution using SA with rough temperature drop for model **II**.

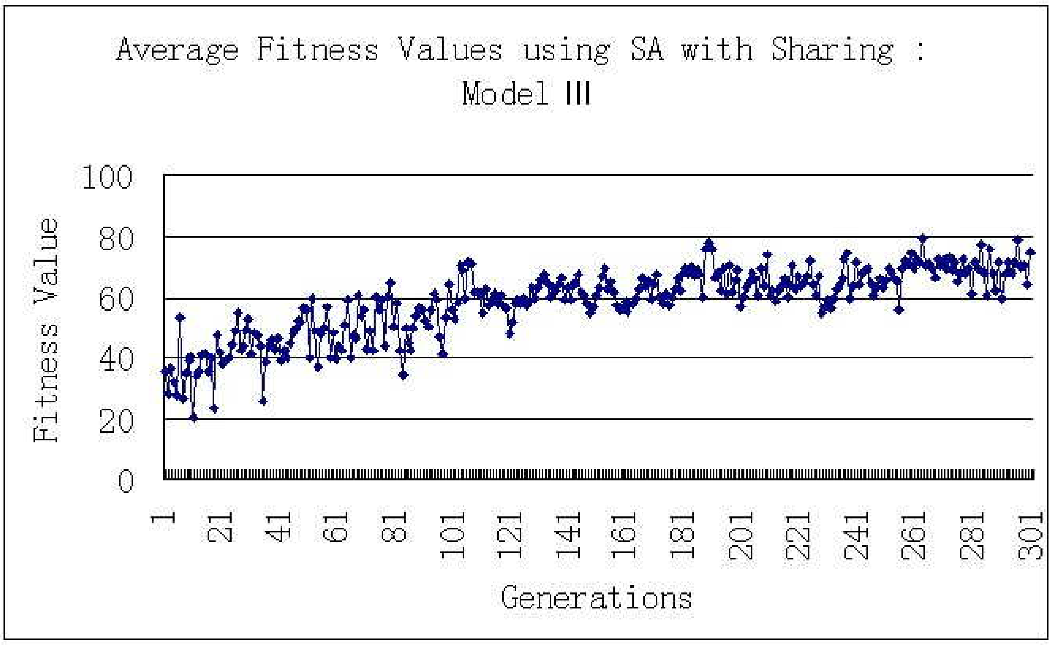


Figure 5.19: Average fitness values when constructing UIOs using SA (rough T drop) for model **III**.

5.6 General evaluation

The experimental results reported in the previous sections suggest that, when constructing UIOs using GA and SA, without sharing technique, the population is likely to converge at several individuals that have high fitness values. The distribution of such a population causes some UIOs to be missed with high probability. After applying sharing technique, the population is encouraged to spread out and forms several sub–populations. These sub–populations are intended to cover all optima in the search space. The search quality significantly improved and more UIOs were found. Tables 5.11 and 5.12 give the number of UIOs that are missed for each experiment using GA, SA, GA with sharing and SA with sharing. From these tables it can be seen that sharing techniques are effective in finding multiple UIOs. It can also be noted that the performance of the search is comparatively stable.

It has also been shown that, with the sharing technique, there is no significant difference between GA and SA on the search for UIOs. Both techniques force their

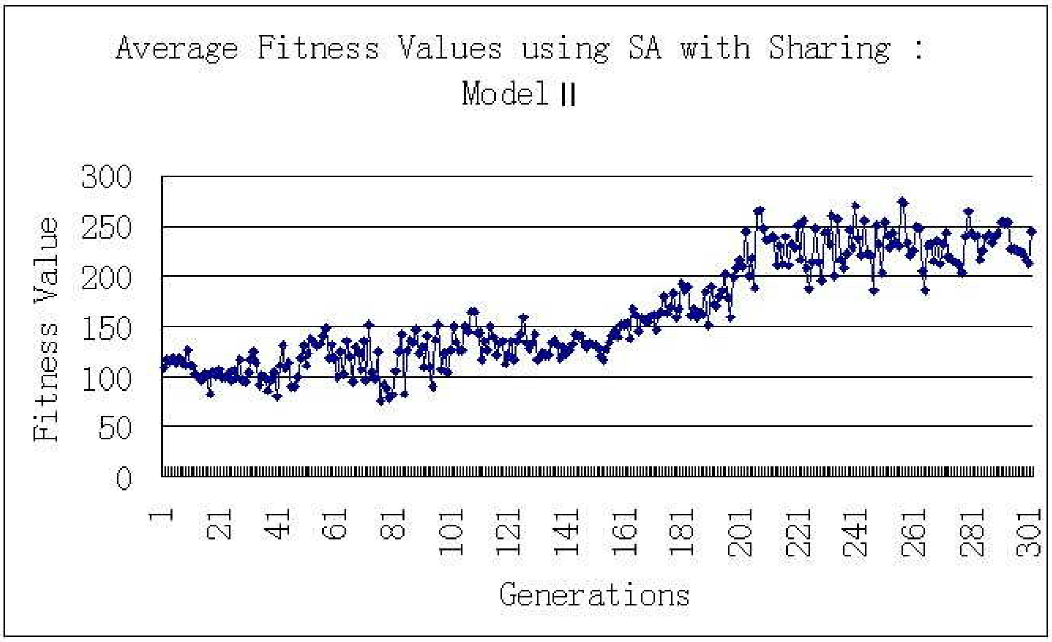


Figure 5.20: Average fitness values when constructing UIOs using SA (rough T drop) for model **II**.

	GA	SA	GA/S	SA/N	SA/R
1	27	56	4	14	2
2	29	49	6	18	4
3	30	62	6	15	4
4	31	37	7	11	3
5	29	55	5	9	6
6	28	48	8	13	5
7	30	44	7	10	2
8	27	51	8	13	6
9	29	39	4	15	4
10	32	46	7	10	3
Avg	29.2	48.7	6.2	12.8	4.1

Table 5.11: Missing UIOs when using model **III**; GA:simple GA without sharing; SA:simple SA without sharing; GA/S:GA with sharing; SA/N:SA with sharing using normal T drop; SA/R:SA with sharing using rough T drop.

	GA	SA	GA/S	SA/N	SA/R
1	7	15	2	3	2
2	7	17	4	4	2
3	9	21	4	3	2
4	7	19	2	3	3
5	7	20	3	3	2
Avg	7.2	18.4	3	3.2	2

Table 5.12: Missing UIOs when using model **II**; GA:simple GA without sharing; SA:simple SA without sharing; GA/S:GA with sharing; SA/N:SA with sharing using normal T drop; SA/R:SA with sharing using rough T drop.

populations to maintain diversity by forming sub–populations. In the two models under test, with the sharing technique, both GA and SA are effective.

When applying the SA, rough exponential temperature drop seems to be better than the normal exponential temperature drop. Since the sharing technique reduces some individuals' fitness values, some information might be lost during the computation. Local bumping in the rough exponential temperature drop gives the experiments a second chance for amendments, which might help to prevent such information from being lost. This could explain why the performance of the rough SA was consistently better than that of a simple SA.

The results on convergence rates imply that, when constructing UIOs, the GA converges faster than the SA. A simple GA works on population based exploration. New solutions (children) inherit information from previous solutions (parents) through crossover while a SA generates a new solution based on the search in the neighbourhood of an existing solution. A simple GA is more exploitative than a SA. That might explain why GA converges faster than SA in our experiments.

5.7 Parameter settings

Parameter settings on crossover and mutation rates follow the suggestions from ref. (Gol89); When investigating the impact of a sharing technique, the population size used in all experiments is fixed to 600. Using a larger size for a population may increase the chance on finding more UIOs, but it increases the computational cost as well. This work did not investigate the effects on varying crossover rate, mutation rate and the population size. Future work will address these issues.

Parameter settings for α and γ affect the performance of computation significantly. γ is defined to control the dynamic behaviour of the exponential part in the fitness function while α adjusts the weight of the linear part. To counteract the effect of $x_i e^{\delta x_i}$, γ must be set to a value that is greater than 1. However, it can also be noted that too big a value of γ causes the calculation of an individual's fitness a continuous decrement even when some discrete partitions are found. Therefore, a comparative small value that is greater than 1 should be considered. In this work, we found that setting γ between 1.2 and 1.5 achieves better performance than other values.

α is defined to reward the partitions when no discrete class has been found. Normally, at the beginning of computation, when no discrete class is found, the *linear part* plays the major role in the calculation of the fitness value. However, with the computation going further and some discrete classes being found, the *exponential part* takes over the role and becomes the major factor. Individuals that switch the role too slowly might obtain low fitness values and become unlikely to be selected for reproduction. This effect might cause some patterns (some UIOs) to be missed.

For example, Figure 5.21 shows two patterns of state splitting trees in model **III**. In pattern **A** (corresponding to *aaaa*), there are 5 discrete units in the fourth layer (corresponding to the first three inputs) and 3 units in the fifth layer (corresponding to the first four inputs). In pattern **B** (corresponding to *ccac*), there is 1 discrete unit in the third layer (corresponding to the first two inputs) and 6 units in the fourth layer (corresponding to the first three inputs). *ccac* acquires a much higher fitness value than that of *aaaa*. *aaaa* is therefore likely to be missed during the computation. To compensate for this effect, a comparatively high α value might be helpful since it enhances the effect of the linear action. In this work, we set α to 20. We have also tested values that are below 15 and found that no experiment discovered the pattern **A** (*aaaa*).

Threshold value θ decides whether the fitness value of a candidate can be reduced. A value between 0 and 1 can be applied. The setting of θ should be

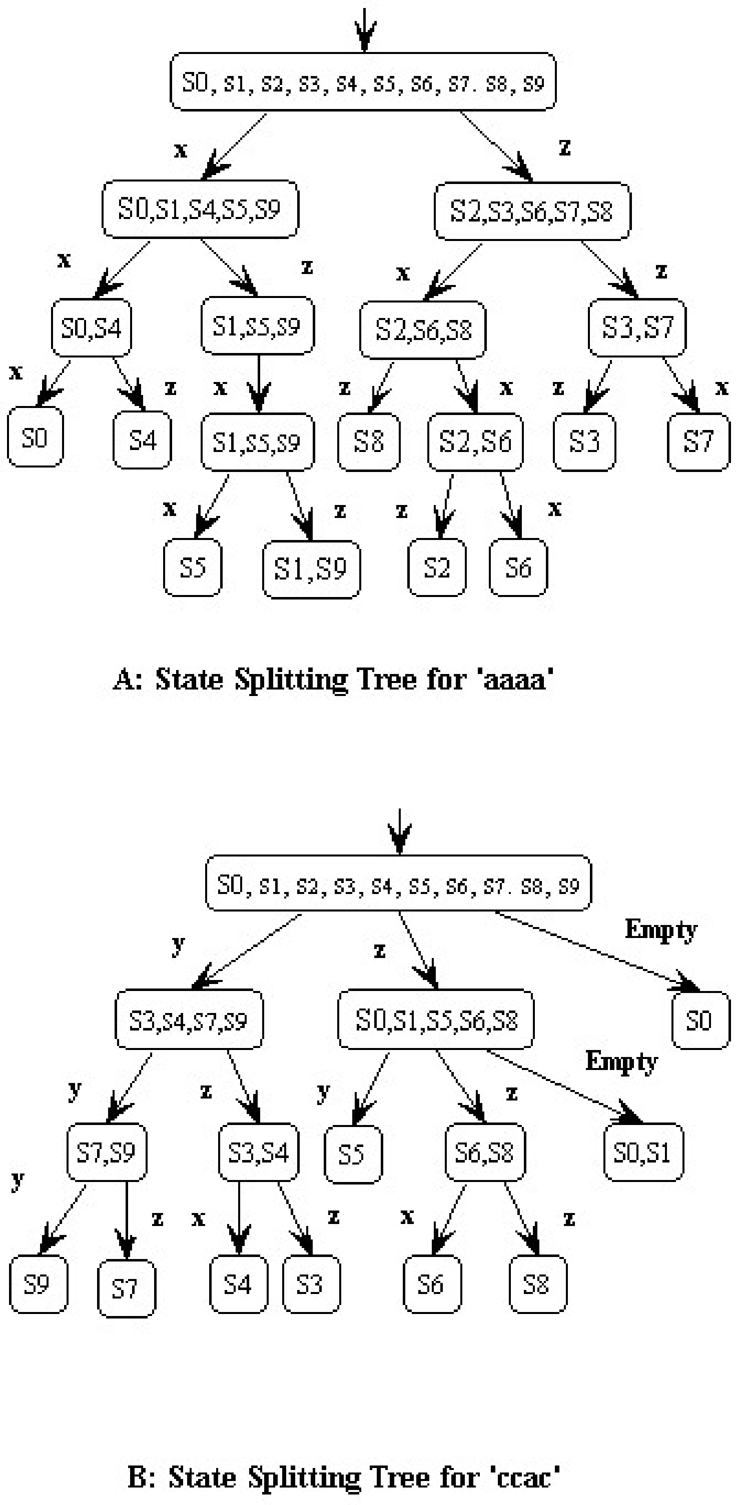


Figure 5.21: Two patterns of state splitting tree generated from model **III**.

suitable and may vary in different systems. If θ is set too low, candidates that are not fully represented by others may be degraded, causing some UIOs to be missed. For instance, *abcab* and *abaac* are two candidates. If θ is set less than 0.4, compared with *abcab*, the fitness value of *abaac* can be degraded. However, it can be seen that *abaac* is not fully represented by *abcab*. *abaac* might be missed in the computation due to inappropriate operations; at the same time, too high a value of θ might make the operation of fitness degrade ineffective. If θ is set to 1, no degrade action occurs.

In our experiments, 2/3 was used in model **III** while 1/2 was selected for model **II**; these values were chosen after some initial experiments.

5.8 Summary

State verification using Unique Input/Output (UIO) sequences has been playing a very important role in the automated generation of test sequences when testing from finite state machines. Finding ways to effectively construct UIO sequences for each state from the finite state machine being investigated is extremely important. However, computing UIO sequences is NP-hard.

In this chapter, we investigated the use of Metaheuristic Optimisation Techniques (MOTs), with sharing, in the generation of (multiple) unique input output sequences (UIOs) from a finite state machine (FSM). A fitness function, based on properties of a state splitting tree, guides the search for UIOs.

The performance of a simple Genetic Algorithm (GA) was experimentally compared to that of random search by using two finite state machines. The experimental results suggested that the GA outperforms random search.

A sharing technique was introduced to maintain the diversity in a population by defining a mechanism that measures the similarity of two sequences. Two finite state machines were used to evaluate the effectiveness of a GA, GA with sharing, and a Simulated Annealing (SA) with sharing. The experimental results showed that, in terms of UIO distributions, there was no significant difference between the GA with sharing and the SA with sharing. Both outperforms the version of the GA without sharing. With the sharing technique, both GA and SA can force a population to form several sub-populations and these are likely to cover many local optima. By finding more local optima, the search identifies more UIOs.

However, a problem is also noted. All sub-populations are dominated by one or several individuals. This remains a research topic for the future work.

Chapter 6

Fault coverage

6.1 Introduction

Testing is an expensive process. Finding effective strategies to automate the generation of efficient tests, which can help to reduce development costs and to improve the quality of (or at least confidence in) a system, is of great value in reducing the cost of system development and testing.

When testing from a finite state machine (FSM) M, an efficient test sequence should cover, as much as possible, all faults which any implementation may have and should be relatively short. Ideally we use a complete test suite: a test suite that is guaranteed to determine correctness if the number of states of the Implementation Under Test (IUT) does not exceed some predetermined bound. However, all approaches to generating a complete test suite either rely on the existence of a distinguishing sequence for M (Gon70; Hen64; UWZ97), assume that the IUT has a reliable reset (Cho78), or produce a test suite whose size is exponential in terms of the number of states of M (RU95). However, a finite state machine need not have a distinguishing sequence and often the IUT does not have a reliable reset and there has thus been much interest in alternative test techniques, often based on UIOs.

Sidhu *et al.* (SL89) concluded that the U-, D-, and W-methods (for more information about these methods, see chapter 4) produce identical fault coverage and ensure the detection of all faults. However, this conclusion was challenged by Chan, arguing that the problems of fault masking in UIOs may degrade the performance of UIO based methods. In their paper (CVI89), Chan *et al.* showed that U-, D- and W-methods produce identical fault coverage only when the UIOs selected from the specification are UIOs in the IUT as well[1]. A UIO may lose its property of uniqueness in some faulty implementations, which leads to the failure of corresponding state verification. To overcome these problems, Chan proposed the UIOv method where all UIOs are checked first for their uniqueness in the IUT before being selected for test case generation. Although this operation helps to improve the test quality, it might significantly increase the test cost. Meanwhile, in a system without a reset function, it might also make the procedure of testing discontinuous.

Naik (Nai95) further studied the problem and pointed out that a fault in a UIO can be masked either by some erroneous outputs or by an incorrect state transfer. In order to enhance the ability of UIOs to resist fault masking, he suggested that, when generating a test sequence those UIOs with maximal strength should be considered first. He also proposed an algorithm to construct UIOs with high strength. This method can effectively reduce the chance that faults are masked by error outputs, but might lack the ability to handle the situation that faults are masked by incorrect state transitions.

Shen *et al.* (SST91) showed that using a backward UIO (B-UIO) in a transition test helps to improve test quality. By applying a B-UIO, the initial state of the transition is also verified. All states in the B-UIO method are verified twice. The test quality is therefore improved. However, the use of B-UIOs can also lead to some problems. These are discussed in section 6.2. In ref. (SL92), Shen and Li extended the work by using Unique Input/Output Circuit (UIOC) sequences for state verification. A UIOC is constructed by using a F-UIO[2] and a B-UIO for a state. If the F-UIO and the B-UIO do not naturally form a circuit (the tail state of the F-UIO is not the initial state of the B-UIO), a transfer sequence will be added to complete it. This operation may give rise to some problems. If the gap between the tail state of the F-UIO and the initial state of the B-UIO is too

[1]Problems described in U-method is suitable for W-method as well where UIO is replaced with CS.

[2]An ordinary UIO is denoted as F-UIO in order to distinguish it from the backward UIO.

long (needs a long transfer sequence), the operation may reduce the robustness of the UIOC for verification.

In this chapter, we investigate the problem of fault masking in UIOs and proposed the use of a new type of UIOC sequence for state verification to overcome the problem. UIOCs themselves are particular types of UIOs where the ending states are the same as their initial states. When constructing a UIOC, by further checking the tail state and by using overlap or internal state observation scheme, the fault types of UIOs discussed in section 6.2 can be avoided, which makes the UIO more robust. Based on rural Chinese postman algorithm and UIOCs for state verification, a new approach is proposed for generating a more robust test sequence.

An approach was also suggested for the construction of B-UIOs. Test performance among F-UIO, B-UIO and UIOC based methods was compared through a set of experiments. The robustness of the UIOCs constructed by the algorithm given in this work and those constructed by the algorithm given in ref. (SL92) were also experimentally compared.

6.2 Problems of the existing methods

6.2.1 Problems of UIO based methods

Unique input/output sequences uniquely identify states in the specification FSM. The UIO based methods are based on the assumption that UIOs in a specification FSM are also UIOs in the IUT. This assumption is however not always true. A faulty example cited from ref. (CVI89) is shown in Figure 6.1. In the specification FSM, sequence $(b/1)(a/1)$ is a UIO for s_3. However, in the faulty implementation, s_1 and s_3 produce the same output (11) when responding to ba. The UIO loses its property of uniqueness in the IUT and fails to identify s_3.

The problem is called *fault masking in UIOs*. The capability of a UIO to resist this problem is called its *strength* (Nai95). In UIO based test methods, the use of UIOs with low strength may lead to a test sequence that is not robust.

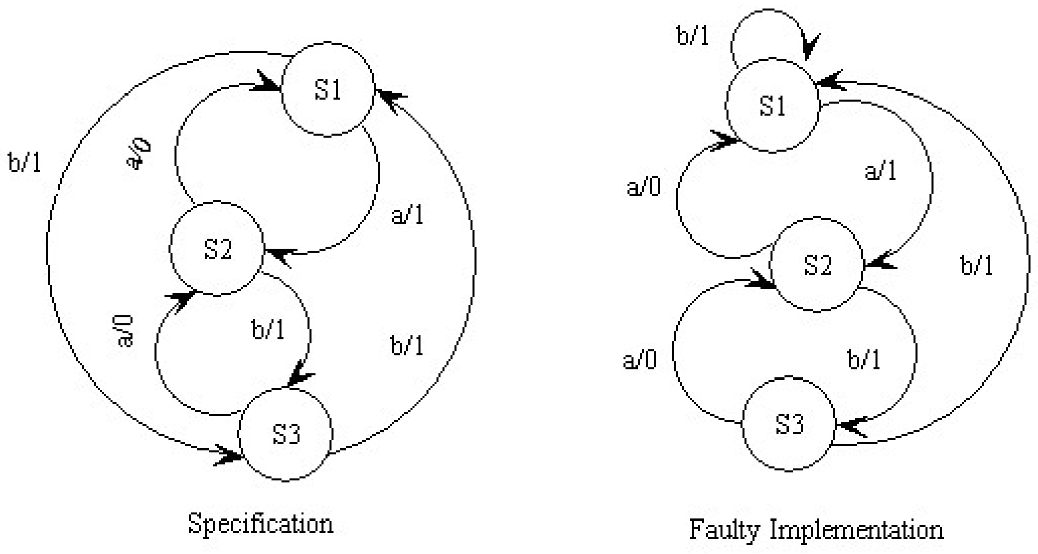


Figure 6.1: A specification finite state machine and one faulty implementation cited from ref. (CVI89).

6.2.2 Problems of backward UIO method

A B-UIO provides evidence that an FSM is currently in a known state, but does not show from which state it initially came. A B-UIO may have several valid initial states that satisfy the definition of this B-UIO. An example is shown in Figure 6.2 where the FSM (table 6.1) is defined in section 6.5. Sequence *dccd* is a B-UIO for s_0. It can be seen that the B-UIO sequence has 4 initial states (s_1, s_4, s_6, s_9) that satisfy the definition. If a B-UIO has more than one valid initial state and is chosen for a transition test, it is possible that a fault that occurred in the previous transition test is masked by this B-UIO.

An example is illustrated in Figure 6.3 where the test segment for transition $s_i \to s_j$ is formed by concatenating the input part of the B-UIO for s_i with the input of this transition and the input part of the F-UIO for s_j. Suppose the B-UIO sequence chosen for s_i has more than one valid initial state while s_m and $s_m^{'}$ are both valid ones and, according to the specification FSM, the previous transition test should end up with s_m. If, in a faulty implementation, the tail state of the previous transition test happens to be $s_m^{'}$, the selected B-UIO for s_i

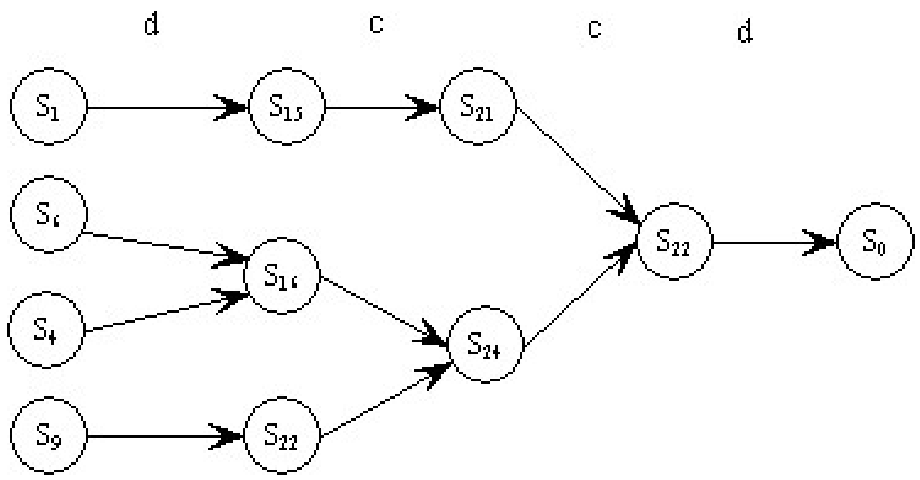


Figure 6.2: "dccd/yyyy": Backward UIO sequence of S_0 in the finite state machine defined in table 6.1.

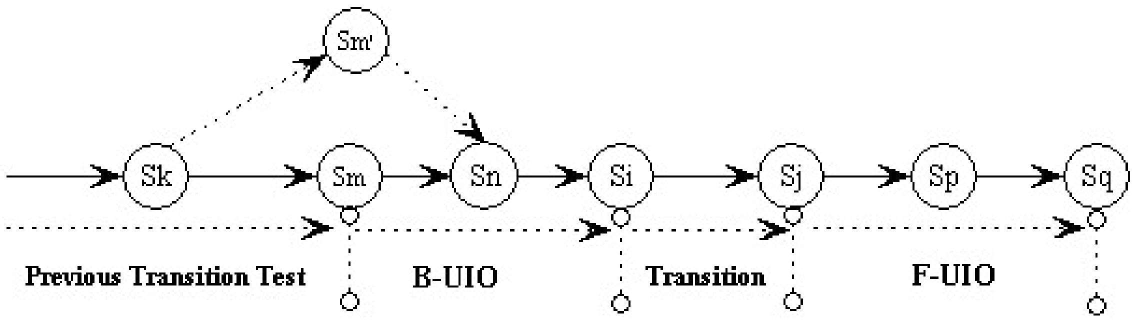


Figure 6.3: Problems of the B-method.

will automatically mask this fault. This makes the test sequence less likely to detect this faulty implementation.

It can be noted that the fault masking problems described in the F-method may happen in the B-method as well since, in the B-method, a transition test consists of a part that uses the F-UIO for the tail state verification. However, since the B-method not only verifies the tail state of a transition, but also checks the initial state, the robustness of a test sequence can be enhanced.

6.3 Basic faulty types

Lombardi *et al.* (LS92) formalise the faulty implementations of UIOs into two basic types shown in Figure 6.4. In this section, we discussed the problems defined by Lombardi *et al.* and proposed solutions to overcome them.

In type 1, the tail state of the UIO in the implementation is different from that in the specification while in type 2 the UIO and its faulty implementation have an identical ending state. The following explains how the faults are masked. Suppose $i_1 i_2 i_3 i_4$ is the input sequence from a UIO for s_i. When the FSM is in s_i and receives $i_1 i_2 i_3 i_4$, it produces $o_1 o_2 o_3 o_4$, visiting s_j, s_k, s_m and s_n correspondingly. In type 1, a fault is caused by an erroneous state transfer $s_i \longrightarrow s_j^{'}$ that has the same I/O as $s_i \longrightarrow s_j$. Instead of being in s_j, the FSM arrives at $s_j^{'}$. If the following outputs are $o_2 o_3 o_4$, these outputs are then masking the state transfer fault. Suppose the following transition test is for a transition $T : s_n \longrightarrow s_x$. If there exists another transition $T^{'} : s_n^{'} \longrightarrow s_x$ that has the same I/O behaviour as T, and, rather than T, the transition $T^{'}$ is tested, the test sequence will therefore be unable to detect the fault in T.

In type 2, an erroneous transition $s_i \longrightarrow s_j^{'}$ that has the same I/O as $s_i \longrightarrow s_j$ occurs. This fault is masked first by an output o_2 and then by another erroneous transition $s_k^{'} \longrightarrow s_m$ that has the same I/O behaviour as $s_k \longrightarrow s_m$.

It can also be noted that the faulty implementations described in F-UIOs can be extended to B-UIOs by considering the tail states as initial states.

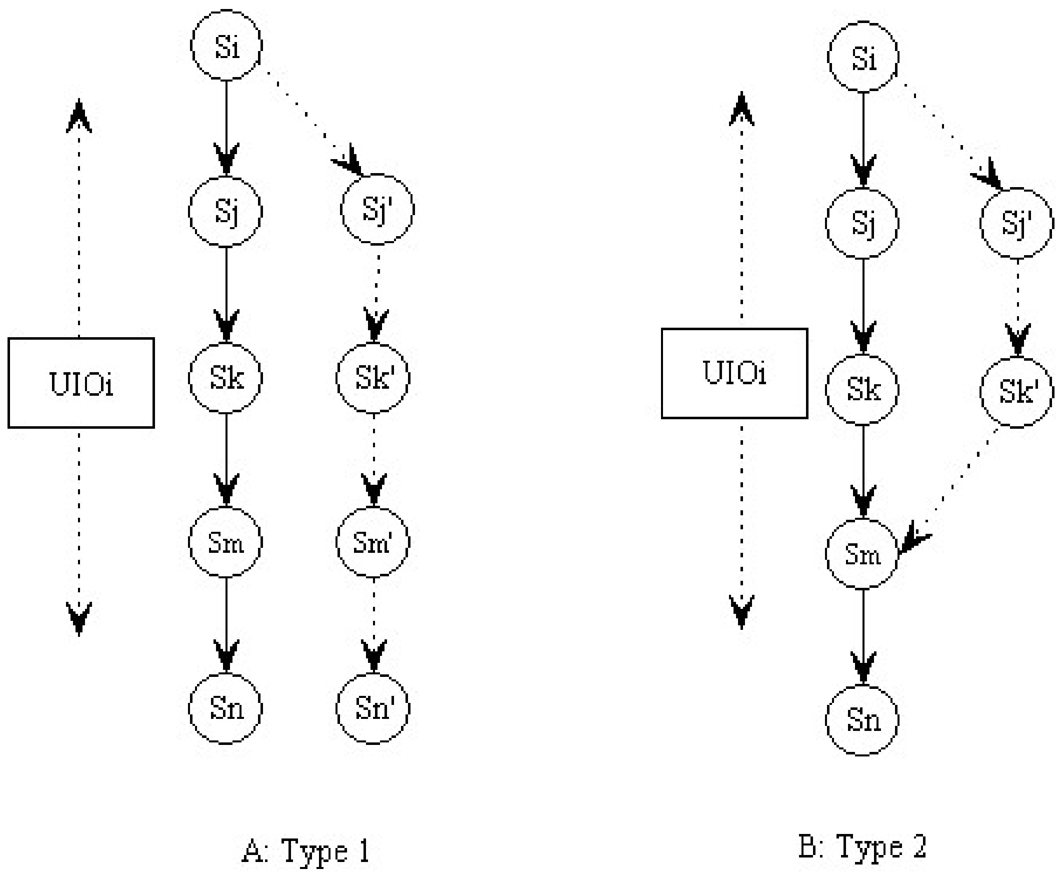


Figure 6.4: Types of faulty UIO implementation.

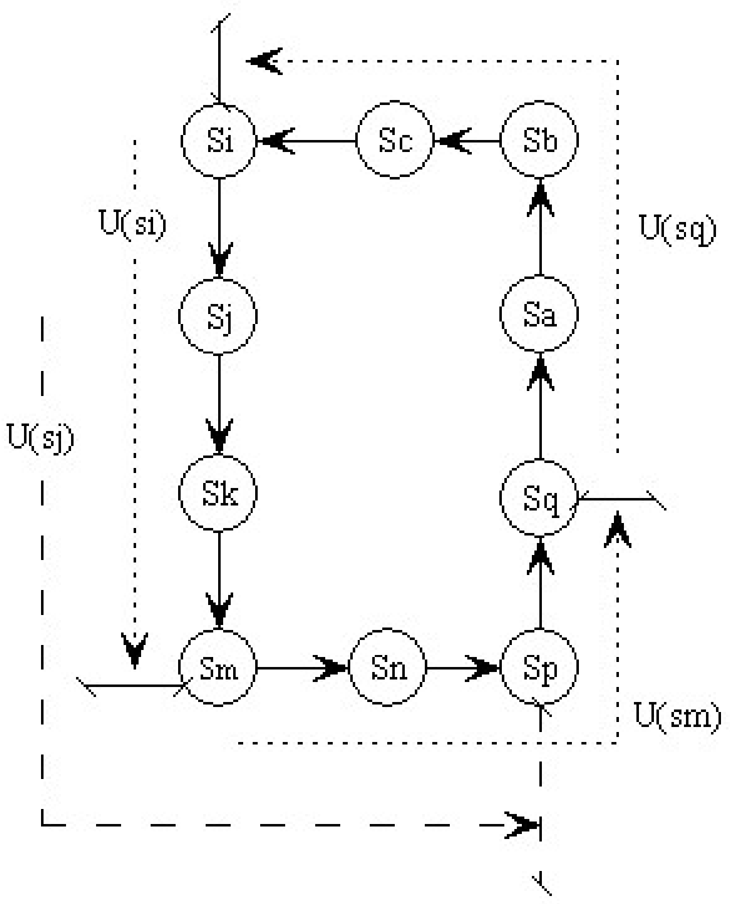


Figure 6.5: Construction of UIOC sequences using overlap scheme.

6.4 Overcoming fault masking using robust UIOCs

6.4.1 Overcoming type 1

In type 1, the final state of a faulty UIO is different from that of its specification, and so one way to detect this error is to further verify it. This is illustrated in Figure 6.5. Suppose, according to its specification, an FSM should be in s_m after applying a UIO sequence U_{s_i} for s_i. To check whether the FSM is in s_m, a UIO sequence U_{s_m} for s_m is then applied, moving the FSM to s_q. If the input/output behaviour is identical to that described in the specification, it then provides evidence that the final state of U_{s_i} (s_m) is correct. A question then arises: how can we be sure that the FSM is in s_q? A UIO sequence for s_q can be used to further check it. The procedure of repeating the verification for the final state of a UIO gives evidence that all previous UIO sequences make the FSM arrive at correct final states. However, the procedure of verifying the final states of UIO sequences should terminate. UIOs should construct a Unique Input/Output Circuit (UIOC) to terminate the verification. The following will give a detailed explanation of the control scheme (Figure 6.5).

Suppose U_{s_i} is a F-UIO for s_i and its final state is s_m. By applying a F-UIO U_{s_m} for s_m, evidence is given, indicating that the FSM was previously in s_m. Continuing to apply F-UIO U_{s_q} for s_q, the I/O behaviour therefore provides evidence that the FSM arrived at s_q. Suppose that there exists a F-UIO U_{s_q} for s_q with tail state s_i, the application of U_{s_q} provides evidence for the correct arrival at s_q. The structure of the UIOC shows that, if the input sequence is executed more than once, the I/O behaviour of U_{s_i} will repeat, which provides evidence for the correct arrival of s_i. Thus, by constructing a UIOC, each UIO provides evidence of the correct arrival of its previous UIO's tail state.

The control scheme shown in Figure 6.5 is an ideal situation. In some applications, the complete UIOC may not be constructed. For example, instead of terminating at s_i, the last UIO sequence in the UIOC may make the FSM arrive at s_c, forming a gap between the tail state and s_i. When dealing with this situation, a shortest sequence can be considered since the extended sequence of a UIO for a state is still a UIO. Meanwhile, when constructing a UIOC, UIOs that result in a minimal gap between the tail state of the last UIO and s_i need to be considered. For instance, if there are two sets of UIOs where the first set moves the FSM to s_c while the other moves the FSM to s_b, the first set of UIOs is better than the other if the gap between s_c and s_i is shorter than that between s_b and s_i.

The UIOC can be constructed by using B-UIOs as well. Suppose, in Figure 6.5, U_{s_i}, U_{s_m} and U_{s_q} are B-UIOs for s_m, s_q and s_i correspondingly, then U_{s_q} provides evidence that the FSM is in s_i (U_{s_i} starts from s_i), U_{s_m} provides evidence that the FSM is in s_q (U_{s_q} starts from s_q) and U_{s_i} provides evidence that the FSM is in s_m (U_{s_m} starts from s_m). Therefore, in a UIOC constructed by B-UIOs, each B-UIO provides evidence for the next B-UIO's initial state.

The advantages of using B-UIOs are that: 1. In a deterministic FSM, each state has at least one B-UIO; 2. A minimal FSM with n states has a homing sequence of length $O(n^2)$ that can be constructed in time $O(n^3)$(Koh78), and B-UIOs can be derived from homing sequences by concatenating the input part of the homing sequence with a transfer sequence that moves the FSM from the tail state of a homing sequence to the target state.

Proposition 6.4.1 *Given a deterministic, reduced and strongly connected finite state machine M, there exists at least one B-UIO sequence for each state of M.*

Proof: In a minimal deterministic FSM, there exists at least one homing sequence H (Koh78). Suppose, when responding to H in some state s, the FSM ends up at state s_i and produces $H_{(o)}$, $H/H_{(o)}$ is a B-UIO for s_i. Given a state s_j, $s_i \neq s_j$, there exists an I/O sequence $L/L_{(o)}$ that moves the FSM from s_i to s_j since the FSM is strongly connected. I/O sequence $HL/H_{(o)}L_{(o)}$ is a B-UIO for s_j. □

Proposition 6.4.2 *In a deterministic finite state machine, if an I/O sequence L_i/L_o is a F-UIO for s_i such that $s_j = \delta(s_i, L_i)$ and $L_o = \lambda(s_i, L_i)$, then L_i/L_o is also a B-UIO for s_j with one valid initial state.*

Proof: Suppose L_i/L_o is a F-UIO for s_i and, when responding to L_i, the FSM arrives at s_j. Since the FSM is deterministic, s_j is the only state reachable by L_i/L_o from s_i. Since L_i/L_o is the F-UIO for s_i, L_i/L_o is a B-UIO for s_j with one valid initial state. □

However, the UIOCs constructed by a complete set of B-UIOs may not be UIOs (F-). Therefore, before choosing the UIOCs for state verification, the uniqueness of the I/O sequences need to be checked. Only input/output sequences that form F-UIOs are used.

A UIOC can be constructed by using both F-UIOs and B-UIOs. It can be seen that the sequence formed by concatenating a F-UIO (head state is s_F) with a B-UIO (tail state is s_B) is a F-UIO for s_F and a B-UIO for s_B. In ref. (SL92) a UIOC was also used for state verification. A UIOC was constructed by using a F-UIO and B-UIO for a state. If the F-UIO and B-UIO do not naturally form a circuit (the tail state of the F-UIO is not the initial state of the B-UIO), a short sequence is applied to complete it. It can be noted that, if the F-UIO and B-UIO form a circuit, the UIOC in (SL92) is identical to that in this work. The construction of UIOCs in (SL92) can be viewed as a special case of this work. However, if there exists a gap between the tail state of the F-UIO and the initial state of the B-UIO, a short sequence has to be applied to complete the circuit. This may degrade the robustness of the UIOC.

This work proposed a new method for generating UIOCs where the F-UIO and the B-UIO do not form a circuit. In section 6.5, we reported the result of an experiment devised to compare the relative robustness of UIOCs that were constructed by the methods given in this work and in ref. (SL92). The experimental results showed that the UIOCs constructed according to the algorithm given in this work are more robust than those constructed by the algorithm given in ref. (SL92).

The use of UIOCs for state verification will increase the length of a test sequence. When constructing a UIOC, if there exists more than one set of F-UIOs or B-UIOs that can form UIOCs, the set with the least number of elements should be used to get a UIOC with the minimal length. Meanwhile, overlap among UIOs needs to be considered as well to further reduce the length.

6.4.2 Overcoming type 2

A: Overlap scheme

In type 2, a transfer fault may be masked by another transfer error. When constructing a UIOC, the consideration of overlap among UIOs for internal states can help to overcome this problem. For example, in Figure 6.5, U_{s_i}, U_{s_m} and U_{s_q} form a UIOC for s_i. If there exists U_{s_j} in the circuit that is a UIO for s_j, the chance that the UIOC fails to find type 2 fault can be reduced. If F-UIOs for all internal state's UIOs, say s_j, s_k and s_m, are included in the UIOC, then the chance to fail to detect type 2 will be reduced.

B: Internal state sampling scheme

When constructing a UIOC for a state, internal states may not be verified by their UIOs in the UIOC sequence. An alternative way to overcome type 2 is

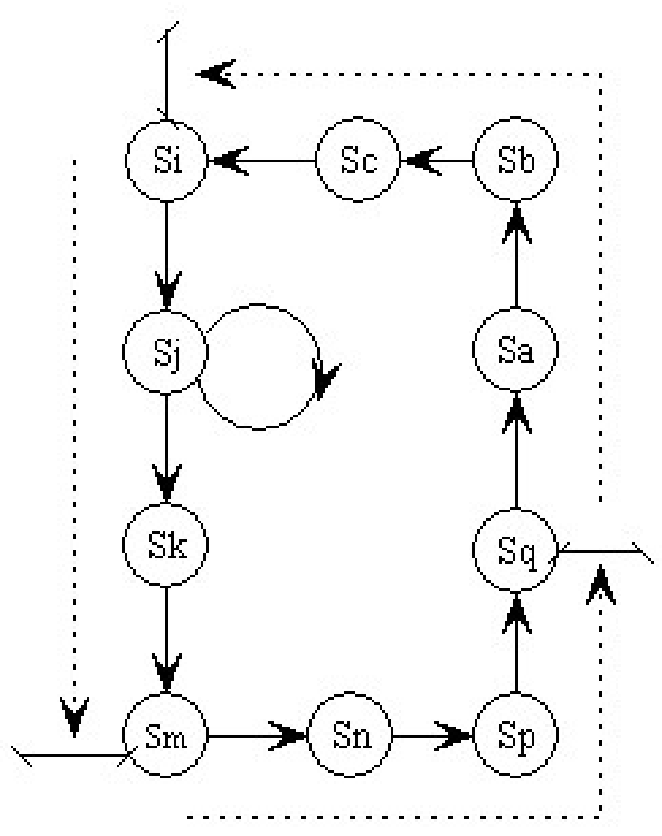


Figure 6.6: Construction of UIOC sequences using internal state sampling scheme.

then to check internal states by adding additional observers that are self-loops. Figure 6.6 shows the scheme where s_j is checked. In a faulty implementation, an observer may either make the FSM produce an erroneous output or arrive at an erroneous state that may be detected by the following verification. This helps to increase confidence that UIOCs constructed from the specification FSM remains UIO (F-) in the IUT. Ideally, the observers are F-UIOs that are naturally loops. But if the F-UIO is too long, a shortest loop sequence could be substituted for the function.

Definition 6.4.1 *A loop sequence LS_i/LS_o for s_i is an I/O sequence such that $s_i = \delta(s_i, LS_i)$ and $LS_o = \lambda(s_i, LS_i)$.*

It would make the verification more robust if all internal states are checked by their observers. However, this will make the test sequence very long. Thus, instead of checking all, one state is selected for verification by the scheme shown in Figure 6.7. Suppose input sequence U_{s_i} is a F-UIO for state s_i. When responding to U_{s_i}, the FSM produces an output sequence O_{s_i} and gives a trace of s_j, s_k, s_l, $s_i^{'}$. Putting U_{s_i} to all other states, we get output sequences and traces correspondingly. Suppose, by comparing the output sequences, a common I/O area is found shown between two dotted lines. The area is said to be a highly dangerous area. A state transfer error is likely to be masked in such an area. For example, $s_i \rightarrow s_j$ might be replaced by $s_i \rightarrow s_p$. This mistake might be masked later by $s_x \rightarrow s_i^{'}$ or by $s_q \rightarrow s_l$. A state between s_j and s_l needs to be further checked by its observer. The middle state, namely s_k, is considered.

6.4.3 Construction of B-UIOs

B-UIOs might be considered for the construction of UIOCs. However, there is no complete algorithm to construct B-UIOs in the literature. Although homing sequences can be used as the basis for the construction, the B-UIOs obtained might be long, which will increase the cost in the forthcoming test. Based on the studies of state splitting tree (see chapter 4), we proposed the State Merging Tree (SMT) for the construction of B-UIOs.

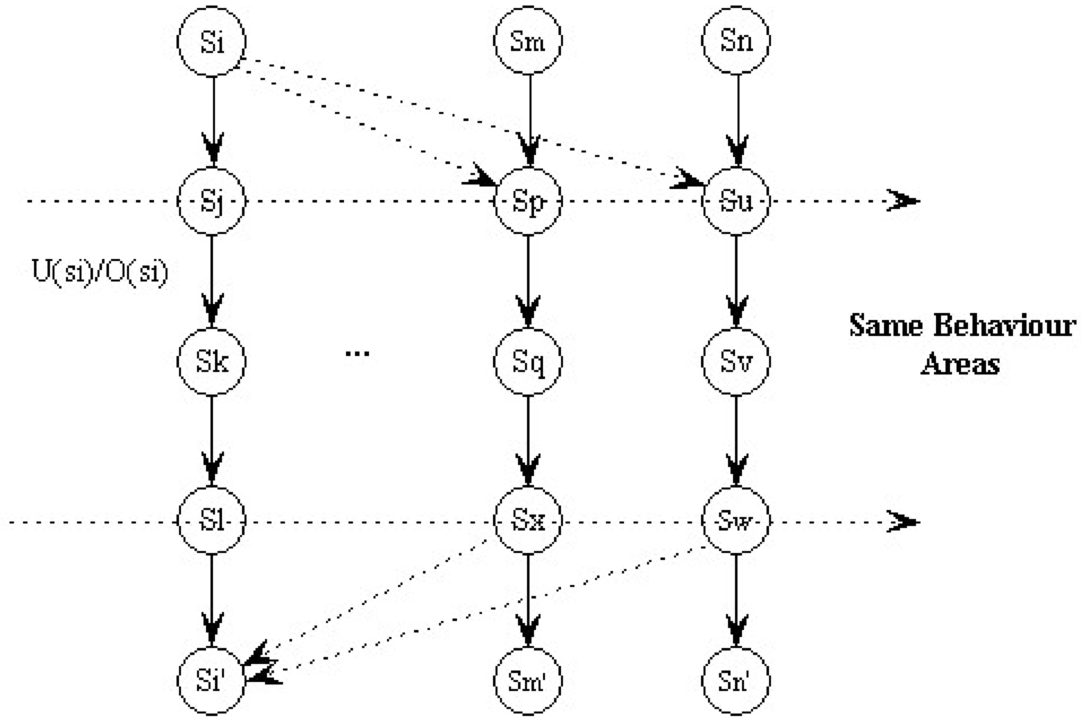


Figure 6.7: Rule on selection of a state.

Similar to a SST, a SMT is a rooted tree. Each node in a SMT contains a set of states where the root node contains the complete set of states and the discrete nodes (terminals) contain one state. A node is connected to its parent by an edge labelled with characters, indicating the situation of state merging. However, differences exist between SST and SMT where at each single input stage, the SST only cares about the initial state from which the current state came while the SMT takes not only the initial state, but also the current state into account.

To give a further explanation, an example is shown in Figure 6.8 where the FSM has 6 states, the input set is $\{a, b\}$, and the output set is $\{x, y\}$. Suppose, when responding to a, $\{s_1, s_3, s_5\}$ produce x and arrives at $\{s_2, s_2, s_3\}$[1], while

[1]Both s_1 and s_3 arrives at s_2. The set of final states is actually $\{s_2, s_3\}$. However, to make

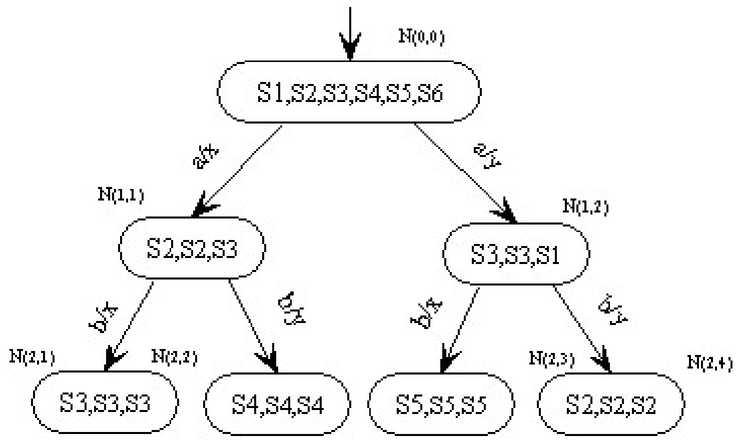


Figure 6.8: The pattern of a state merging tree from an FSM .

$\{s_2, s_4, s_6\}$ produce y and arrive at $\{s_3, s_3, s_1\}$. a is said to merge s_1 and s_3 at s_2 by producing x and to merge s_2 and s_4 at s_3 by producing y. Two nodes are generated from the root node indicated by $N(1,1)$ and $N(1,2)$. Continuing to input the FSM with b, if states reached from $\{s_1, s_3, s_5\}$ by a arrive at $\{s_3, s_3, s_3\}$ producing x or arrive at $\{s_4, s_4, s_4\}$ producing y, ab is said to merge $\{s_1, s_3, s_5\}$ at s_3 by producing xx or $\{s_1, s_3, s_5\}$ at s_4 by producing xy. Two nodes rooted from $N(1,1)$ are then generated indicated by $N(2,1)$ and $N(2,2)$. Once a discrete node such as $N(2,1)$ occurs, the path from the root node to $N(2,1)$ forms a B-UIO for the corresponding state (in this case, s_3). By the nature of the tree, s_3 is the only state that can be reached by this input/output sequence. If all of the terminal nodes are discrete nodes and all the input sequences defined by paths from the root node to terminal nodes are the same input sequence x then x is a homing sequence.

It can be seen that one SMT may not contain B-UIOs for the complete set of states. It may be necessary to generate several SMTs to provide the B-UIOs for every state. Once the SMTs are defined, the model for the construction of UIOs proposed in chapter 5 can be extended for the construction of B-UIOs.

the explanation clearer, all final states remain listed.

6.4.4 Construction of UIOCs

The construction of UIOCs in this work follows three schema: 1. A complete set of F-UIOs was used; 2. A mixed set of F-UIOs and B-UIOs was used. 3. A complete set of B-UIOs was used. When constructing a UIOC, the first scheme is considered. If a UIOC cannot be constructed by a complete set of F-UIOs, the second scheme will be considered. Only when the first and the second schema fail to construct a UIOC, will the third scheme be considered. All three schema should take the overlap or the internal state observation scheme into account. When the internal state observation scheme is used, a self-loop F-UIO (if the observed state has one) is first considered. However, if the F-UIO is too long, a loop sequence is substituted for the function. In this work, if the length of a self-loop F-UIO of a state is greater than 4, a shorter loop sequence is then used for the observation of this state.

When the B-UIOs are used for the construction of a UIOC, those B-UIOs with fewer number of valid initial states should be used to avoid the fault masking problem caused by B-UIOs. When a UIOC is constructed by a complete set of B-UIOs, the UIOC may not be a UIO (F-).

Before a UIOC is selected for state verification, its uniqueness needs to be checked to make sure that it is a F-UIO.

6.5 Simulations

A set of experiments was devised to compare the test performance among F-, B-, and C-Method. In all our experiments, we used FSMs where the size of the state set and the input set are much higher than that of the output set. The structure of the FSMs could make testing harder. We believe that, in these kinds of FSMs, the UIOs for each state tend to be long and the problems of fault masking are likely to happen.

A randomly generated FSM is first defined in [1]Table 6.1.

The system has 25 states while the input set is $\{a, b, c, d\}$ and the output set is $\{x, y\}$. The FSM is reduced, deterministic and completely specified. There

[1]Contents in the first row are inputs. s_i: s_j/y means that, when the FSM is in s_i and receives an input shown in the first row, it moves to s_j and produces y.

	a	b	c	d
S0	S3/y	S12/y	S10/x	S24/x
S1	S4/x	S11/x	S23/y	S15/y
S2	S17/x	S9/x	S14/y	S0/y
S3	S5/x	S0/y	S13/x	S23/y
S4	S20/x	S18/x	S15/y	S16/y
S5	S3/y	S1/x	S12/x	S20/y
S6	S1/y	S7/x	S19/x	S16/y
S7	S21/x	S24/y	S9/y	S6/x
S8	S14/y	S12/y	S18/x	S5/x
S9	S15/x	S2/y	S6/y	S22/x
S10	S11/x	S19/y	S23/y	S8/x
S11	S17/x	S12/x	S0/y	S6/y
S12	S9/x	S13/y	S20/x	S1/y
S13	S7/x	S4/y	S10/x	S22/y
S14	S8/x	S3/y	S19/y	S11/x
S15	S6/x	S17/x	S21/y	S2/y
S16	S7/y	S20/y	S24/x	S4/x
S17	S15/y	S13/x	S2/x	S8/y
S18	S16/x	S5/y	S20/x	S10/y
S19	S23/x	S11/y	S9/y	S18/x
S20	S14/y	S21/x	S17/y	S7/x
S21	S4/x	S16/x	S22/y	S1/y
S22	S10/x	S2/x	S24/y	S0/y
S23	S18/y	S21/x	S13/x	S3/y
S24	S14/y	S5/y	S22/x	S8/x

Table 6.1: Specification finite state machine with 25 states used for simulations.

Methods	Specification	Mutants
F-Method	$s_4 - (a/x) \rightarrow s_{20}$	$s_4 - (a/x) \rightarrow s_7$
	$s_5 - (a/y) \rightarrow s_3$	$s_5 - (a/y) \rightarrow s_{23}$
	$s_{24} - (d/x) \rightarrow s_8$	$s_{24} - (d/x) \rightarrow s_0$
B-Method	$s_{18} - (c/x) \rightarrow s_{20}$	$s_{18} - (c/x) \rightarrow s_8$
	$s_{18} - (c/x) \rightarrow s_{20}$	$s_{18} - (c/x) \rightarrow s_{24}$
	$s_{23} - (d/y) \rightarrow s_3$	$s_{23} - (d/y) \rightarrow s_{23}$

Table 6.2: Examples of faulty implementations that F- and B-method fail to detect.

are $4 \times 25 = 100$ transitions. A mutant (faulty implementation) is generated by modifying a transition. The selected transition is changed either on its output or the final state. There are $100 \times 24 + 100 = 2,500$ mutants. Test sequences are generated with the F-, the B- and the (new) C-method separately and then used to check all these mutants. To make the explanation clear, the test sequences generated with the F-, the B- and the C-method are called F-, B- and C- sequence correspondingly. In the experiment, we found that 199 mutants passed the F-sequence, 3 passed the B-sequence and none of them passed the C-sequence. This result suggests that the B-method is better than the F-method, which is consistent with the work of (SST91).

However, there are still 3 mutants that passed the B-sequence but were found by the C-sequence. Six examples of faulty implementations where 3 passed the F-sequence and 3 passed the B-sequence are shown in Table 6.2. This result suggests that the test sequences generated with the C-method are more robust than that produced using the B-method. We also compared the lengths of the test sequences. They are 506 (F-sequence), 915 (B-sequence) and 1015 (C-sequence). Compared to the F-method, the B-method increases the length roughly by 45% while the C-sequence is approximately 10% longer than the B-sequence. We then increased the length of the F-sequence and the B-sequence to 1015 by adding input characters that were randomly selected from the input set. The experiment was repeated 10 times and the best result was selected. The final result showed that 89 mutants passed the extended F-sequence, and both the extend F- and B-sequences failed to find the three mutants that passed the test in the previous

FSM	F-Num	C-Num
$s_{18} - (c/x) \rightarrow s_{20}/s_8$	3	0
$s_{18} - (c/x) \rightarrow s_{20}/s_{24}$	3	0
$s_{24} - (d/x) \rightarrow s_8/s_0$	5	0
$s_5 - (a/y) \rightarrow s_3/s_{23}$	5	0
$s_5 - (c/x) \rightarrow s_{12}/s_{18}$	5	0

Table 6.3: Numbers of F-UIOs and UIOCs that lost the property of uniqueness in the faulty implementations.

experiment. This experiment suggested that, although extending the F-sequence to a certain length may help to improve its ability to find more errors, it is still less robust than the B-sequence and the C-sequence.

The numbers of F-UIOs and UIOCs that lost the property of UIOs in the faulty implementations was also studied. Five mutants were chosen for the experiment. Test results are shown in Table 6.3 where $s_{18} - (c/x) \rightarrow s_{20}/s_8$ indicates that a mutant was generated by changing the final state s_{20} to s_8 while F-Num and C-Num show the numbers of F-UIOs and UIOCs that are no longer UIOs in the faulty implementations. From the table it can be seen that no UIOCs lost the property of UIOs but some F-UIOs did. These results suggest that the UIOCs provided by the algorithm given in this work are more robust than F-UIOs.

An experiment was designed to compare the UIOC (constructed by the algorithm in this work) with the F-UIO that are constructed by two shortest F-UIOs where one is used to verify the state under test while the other to verify the tail state of the previous F-UIO. Experimental result showed that 47 mutants passed the sequence generated with the latter scheme. Comparing to the F-sequence and the randomly extended F-sequence, the test sequence generated by using two F-UIOs finds more faults, but is still less robust than the test sequence generated with the C-method.

Next, the robustness of UIOCs that were constructed with different schema was compared. Four sets of UIOCs were used. One set was constructed by F-UIOs or B-UIOs, taking the overlap scheme or the internal state observation scheme into account. When only B-UIOs were used to construct a UIOC, the uniqueness of the UIOC was checked to make sure that it is a UIO (F-); one set was constructed by a complete set of F-UIOs, without using the overlap or the internal state observation scheme; one set was constructed by using F-UIOs and B-UIOs that can naturally form circuits; the last was constructed by using F-UIOs and B-UIOs that cannot form circuits. A set of transfer sequences was added to complete the circuits. The experimental showed that 4 mutants passed the UIOC sequence generated using F-UIOs, B-UIOs and transfer sequences; 1 passed the UIOC sequences generated using a complete set of F-UIOs without using overlap or internal observation scheme; 1 passed the UIOC using F-UIOs and B-UIOs without using overlap or internal state observation scheme. The latter two sequences failed to find the same mutant. It can be seen that test sequences generated using F-UIOs, B-UIOs and a set of transfer sequences showed even worse test performance than those generated with B-method. The experimental results suggest that UIOCs constructed by the algorithm given in ref. (SL92) (using a transfer sequence to complete the circuit) are less robust than those that were constructed by the algorithm given in this work. Compared to the test results of the corresponding test sequences, it can be suggested that the use of the overlap or the internal state observation scheme is likely to make the UIOCs more robust.

We also investigated the test performance of the F-UIO, B-UIO and UIOC methods when applied to FSMs with different numbers of states. All FSMs are randomly generated. They are completely specified, deterministic and strongly connected. The input set and the output set for all FSMs are $\{a, b, c, d\}$ and $\{x, y\}$. All UIOCs constructed by a complete set of B-UIOs are verified to be F-UIOs. The result of the experiment is shown in Table 6.4. The table shows no significant relationship between the number of states and the number of mutants that passed the test, which indicates that the quality of testing is not only determined by the test method, but also determined by the structure of systems. But it can still be seen that, for all FSMs tested, the test sequence generated with the C-method is better than or equal to others. In the experiments, there are $20 \times 4 + 20 = 100$, 400, 900, 1600, 2500, 3600, 4900 and 6400 mutants in the FSMs with 5, 10, 15, 20, 25, 30, 35 and 40 states respectively. Therefore, the total number of mutants in the experiments is 30390. Three mutants passed the C-sequences, which implies that C-method achieves 99.99% fault coverage in the experiments.

States	F-method	B-method	C-method
5	0	0	0
10	1	1	1
15	1	1	0
20	17	8	0
25	199	3	0
30	0	0	0
35	156	8	1
40	9	2	1

Table 6.4: Mutants that pass the test.

The mutant that passed the C-sequence in the FSM with 10 states was studied. In the implementation, transition tr_{14}=$s_3 - (c/y) \rightarrow s_4$ is mutated by changing s_4 to s_3. All UIOC sequences generated from the specification FSM were then checked for the uniqueness in the IUT. When applying the input part of the UIOC sequence for s_4, we found that both s_3 and s_4 produced the same output. Thus, UIOC sequence for s_4 loses the uniqueness in the IUT and fails to identify s_4. Figure 6.9 shows the sequences of transitions traversed by this UIOC sequence in the specification FSM and the faulty implementation respectively. By examining the structures of all UIOC sequences, we found that none of the UIOC sequence traverses transition tr_{14}. This determines that the faulty implementation of tr_{14} is less likely to be detected in the stages of internal state verification, which reduces the chance on finding this error. From figure 6.9 it can also be noted that the faulty implementation of tr_{14} ends up at the state that is exactly the first state, s_3, that the UIOC sequence traverses when it is applied to verify s_4. If the faulty implementation of tr_{14} ends up at another internal state of the UIOC sequence, the fault may be detected by the internal state verification. However, the fault occurs before the process of internal state verification starts. Since tr_{14} holds the same I/O behaviour as that of transition $s_4 - (c/y) \rightarrow s_3$, the fault is likely to be masked in the C-sequence. This work did not provide solutions to overcome the problem. Future work will address this issue.

The lengths of test sequences generated with F-, B- and C-methods were also compared. Table 6.5 shows the lengths of test sequences for different systems. It

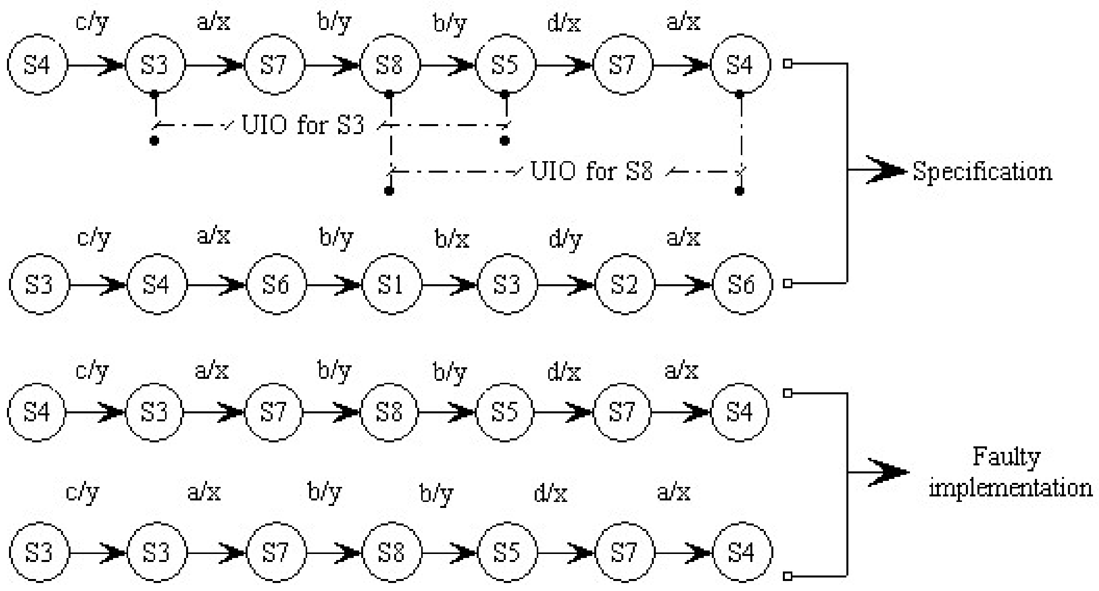


Figure 6.9: UIOC sequence for s_4 in the FSM with 10 states and the faulty implementation that causes the fault masking in the UIOC sequence.

can be seen that the sequences generated with the F-method are always shorter than that with the B- and the C-method. However, the test sequences generated with the C-method are not always longer than those produced using the B-method. In the majority of studies, the C-sequences were slightly longer than the B-sequence while in some cases such as the FSM with 20 states the C-sequence was shorter than the B-sequence.

6.6 Summary

In this chapter, we investigated the problem of fault masking in UIOs. Based on the work of (LS92), two basic types of fault masking involving UIOs were formalised. A new type of UIOC was proposed to overcome the two fault types. When constructing a UIOC sequence, by further checking the tail state of a UIO, type 1 in the faulty implementation may be avoided while by introducing the overlap scheme and internal state observation scheme, type 2 may be avoided. The procedure of verifying the final states of UIOs was terminated by the con-

States	F-method	B-method	C-method
5	88	160	98
10	198	338	351
15	290	484	502
20	398	687	670
25	506	915	1015
30	684	1137	1123
35	740	1315	1339
40	857	1521	1568

Table 6.5: Lengths of the test sequences.

struction of circuits where every UIO provides evidence for the correctness of the previous UIO's tail state.

A set of experiments was designed to study the test performance. Experimental results showed that many more faulty implementations passed the F-sequence than the B-sequence. This suggested that the B-method was more robust than the F-method, which is consistent with the work in ref. (SST91); meanwhile, in the experiment, no faulty implementation passed the C-sequence, which suggested that the C-method is more robust than the F-method and the B-method.

Performance of UIOCs constructed by the algorithm given in this paper and in ref. (SL92) was also compared. Experimental results showed that UIOCs constructed by the algorithm given in ref. (SL92)(using a transfer sequence to complete a circuit) were less robust than those constructed by the algorithm given in this paper. The experimental results also suggested that the use of the overlap or internal state observation scheme is likely to make the UIOCs more robust. In this work, internal state observation scheme considered the sampling of one internal state of a UIO sequence. If a UIO sequence is comparatively long, in order to increase the test confidence, more than one state might be considered for observation. In future work, more studies will be proceeded to investigate the effectiveness of internal sampling schemes.

A set of FSMs was devised to compare the test performance among different methods. Experimental results showed that the (new) C-method was consistently better than or equal to the F-method and the B-method. In the devised experiments, the (new) C-method achieved more than 99.99% fault coverage. It also showed that the C-sequences were not always longer than the B-sequences. In the majority of studies, the C-sequences were slightly longer than the B-sequence while in some cases, the C-sequences were shorter than the B-sequences.

However, it has also been noted that a few of the faulty implementations passed the C-sequences in the experiments. More work needs to be carried on to study the factors that caused the failure of C-sequences.

Chapter 7

Fault isolation and identification

7.1 Introduction

The process of testing aims to check whether the system being developed confirms to its specification. When testing from finite state machines, a set of test sequences is usually required for conformance testing. These test sequences are applied to the Implementation Under Test (IUT) for fault detection. I/O differences exhibited between the IUT and its specification suggest the existence of faults in the implementation. The first observed faulty I/O pair in an observed input/output sequence is called a *symptom*. A symptom could have been caused by either an incorrect output (an *output fault*) or an earlier incorrect state transfer (a *state transfer fault*). Applying strategies to determine the location of faults is therefore important.

Ghedamsi and Bochmann (GB92; GBD93) generate a set of transitions whose failure could explain the behaviour exhibited. These transitions are called *candidates*. They then produce tests (called distinguishing tests) in order to find the faulty transitions within this set. However, in their approach, the cost of generating a conflict set is not considered. Hierons (Hie98) extended the approach to a special case where a state identification process is known to be correct. Test cost is then analysed by applying statistical methods. As the problem of optimising the cost of testing is NP-hard (Hie98), heuristic optimisation techniques such as Tabu Search (TS) and Hill Climbing (HC) are therefore suggested (Hie98).

This chapter studies fault diagnosis when testing from finite state machines.

The work is motivated by an interesting question described as follows. Let ts be a test sequence of length L and let the i^{th} input of ts, $1 \leq i \leq L$, execute a faulty transition tr_f in the IUT. The question is whether it is possible to define the maximum number of inputs that is needed to reveal the failure (a symptom is exhibited) after tr_f being executed. In other words, given a symptom exhibited at the j^{th} input of ts, is it possible to define an interval with a maximum range d_{max}, $d_{max} \geq 0$, such that inputs between $(j - d_{max})^{th}$ and the j^{th} of ts execute a sequence of transitions that must contain tr_f? If such an interval can be defined, the process of fault isolation is then reduced to that of fault identification in a shorter test sequence.

Obviously, the smaller d_{max} is, the less number of transitions will be considered when isolating the faulty transition. It is always preferred that a symptom is observed immediately after a faulty transition is executed. However, it may require more inputs to exhibit the fault.

Clearly, the sequence of transitions executed up to the symptom contains the faulty transition that causes the occurrence of the symptom. However, diagnosing within such a set of candidates might result in a high cost of fault isolation. Finding ways to define the maximum number of inputs that is required to exhibit an executed fault is therefore of great value for minimising the cost of fault isolation and identification.

In this work, heuristics are proposed for fault diagnosis, which helps to reduce the cost of fault isolation and identification. In the proposed method, a set of transitions with minimum size is constructed to isolate the faulty transition that could explain an observed symptom. The erroneous final state of the isolated faulty transition is further identified by applying the proposed heuristics. The heuristics defined in this work consider the use of the U-method (ATLU91). One can easily extend the approach to other formal methods such as the W-method (Cho78) and the Wp-method (FBK+91).

7.2 Isolating single fault

This section introduces an approach for detecting a single fault in the IUT and the construction of a conflict set for fault diagnosis.

7.2.1 Detecting a single fault

When testing an IUT, a set of tests $TC = \{tc_1, tc_2, ..., tc_l\}$ needs to be developed. A test tc_i consists of a sequence of expected transitions $\langle t_{i,1}, t_{i,2}, ..., t_{i,n_i}\rangle$, starting at s_0, with input $\langle x_{i,1}, x_{i,2}, ..., x_{i,n_i}\rangle$ and the expected output $\langle y_{i,1}, y_{i,2}, ..., y_{i,n_i}\rangle$ where y_{i,n_i} is the expected output after input x_{i,n_i}. When executed, tc_i produces the observed output $\langle z_{i,1}, z_{i,2}, ..., z_{i,n_i}\rangle$. If differences between $y_i = \langle y_{i,1}, y_{i,2}, ..., y_{i,n_i}\rangle$ and $z_i = \langle z_{i,1}, z_{i,2}, ..., z_{i,n_i}\rangle$ appear, there must exist at least one faulty transition in the implementation. The first difference exhibited between y_i and z_i is called a *symptom*. Additional tests are necessary in order to isolate the faulty transitions that cause the observed symptom.

7.2.2 Generating conflict sets

A conflict set is a set of transitions, each of which could be used to explain a symptom exhibited. Here, the work focuses on identifying the faulty transition that is responsible for the first exhibited symptom. The transitions after the symptom are ignored.

Suppose, for a test tc_i, the sequence of expected transitions is $\langle t_{i,1}, t_{i,2}, ..., t_{i,n_i}\rangle$ where n_i is the number of transitions. When executed with $\langle x_{i,1}, x_{i,2}, ..., x_{i,n_i}\rangle$, a symptom occurs at the input $x_{i,l}$, the conflict set of the maximum size is $\{t_{i,1}, t_{i,2}, ..., t_{i,l}\}$ where $1 \le l \le n_i$.

7.3 Minimising the size of a conflict set

If the number of transitions in a conflict set is large, the effort required for isolating the fault could be high. It is therefore useful to reduce the size of a conflict set. Two abstract schema are applied in this work, these being:

1. Transition removals using transfer sequences.

2. Transition removals using repeated states.

In the first scheme, a short transfer sequence is used to remove a segment of inputs from the original test sequence. This may lead to a symptom being observed in a shorter test sequence; while, in the second scheme, a segment of inputs is further removed from the original test sequence. These inputs execute a sequence of transitions where the initial state of the first transition is the final state of the last transition. By such an operation, a symptom might be observed in a shorter test sequence, which helps to reduce the cost of fault isolation. Two removal schema are discussed in the following subsections.

7.3.1 Estimating a fault location

Once a symptom is observed, the set of transitions executed up to the symptom constitutes a conflict set $S_{conflict}$ with the maximum size. A subset of transitions $S_r \subset S_{conflict}$ might be removed to reduce the size of $S_{conflict}$ by applying some transfer sequences. Before explaining this in detail, some concepts are defined.

Definition 7.3.1 *A UIO sequence generated from the specification FSM is a strong UIO if it can identify the corresponding state in the IUT; otherwise, it is a weak UIO.*

Due to the problem of fault masking in UIOs, a UIO sequence generated from the specification FSM might lose its property of uniqueness and fail to identify its corresponding state in the IUT (CVI89; Nai95).

Definition 7.3.2 *When testing an IUT, if the UIOs used for the generation of a test sequence are all strong UIOs, the test is a strong test and the test sequence is a strong test sequence; otherwise, the test is a weak test and the test sequence is a weak test sequence.*

Definition 7.3.3 *In a UIO-based test, if there are k weak UIOs in the test sequence, the test is called a $k-degree$ weak test and the test sequence is a $k-degree$ weak test sequence.*

It can be seen that a strong test is a $0 - degree$ weak test.

Definition 7.3.4 *Let [a,b] be the interval of transitions between the a^{th} and the b^{th} inputs from an input sequence α. A transition tr is said to be within [a,b] of α if the c^{th} input executes tr when α is applied to the FSM for some $a \le c \le b$.*

In FSM-based testing, a complete test sequence should test all transitions in the FSM M. A transition is tested by checking its I/O behaviour plus the tail state verification. Once a transition test is finished, M arrives at a state s. If s is not the initial state $s^{'}$ of the transition selected for the following test, a transfer sequence is required to move the M to $s^{'}$. This transfer sequence constitutes a linking sequence in the final test sequence.

Definition 7.3.5 *A linking sequence in a test sequence for an FSM M is a transfer sequence that moves M to the initial state of a transition under test after the previous transition test is finished.*

Proposition 7.3.1 *In a UIO-based test, if the test is a strong test and a symptom is observed at the* a^{th} *input, then the faulty transition that causes the occurrence of the symptom must be within [(a-*$L_{UIO(max)}$*-*$L_{Link(max)}$*), a] of the inputs where* $L_{UIO(max)}$ *is the max length of UIOs and* $L_{Link(max)}$ *the max length of linking sequences.*

Proof: The standard strategy of a transition test in UIO-based test is formed by a transition I/O test and the tail state verification. Since the test is a strong test, no problem of fault masking exists in the test sequence. Suppose a faulty transition is executed at the b^{th} input and the fault is unveiled with an observable symptom at the a^{th} input. If the transition has an I/O error, the fault is detected by the b^{th} input ($a = b$); otherwise, if the transition is one under test, the faulty final state is detected by the following state verification with maximum length $L_{UIO(max)}$ ($b \leq a \leq b + L_{UIO(max)}$), or, if the faulty transition is an element of a linking sequence, the following inputs move the machine to the initial state of the next transition under test with the max steps of $L_{Link(max)}$. After executing the transition with one input, the faulty final state can be detected by the forthcoming state verification with the max steps of $L_{UIO(max)}$ ($b \leq a \leq b + L_{Link(max)} + 1 + L_{UIO(max)}$). Therefore, if a symptom is observed by a strong test sequence at the a^{th} input, the faulty transition that caused the occurrence of the symptom must be within $[(a - L_{UIO(max)} - L_{Link(max)}), a]$ of the inputs. □

Definition 7.3.6 *In a weak test, if a UIO sequence fails to identify the corresponding state in the IUT more than once, the problem is called fault masked UIO cycling.*

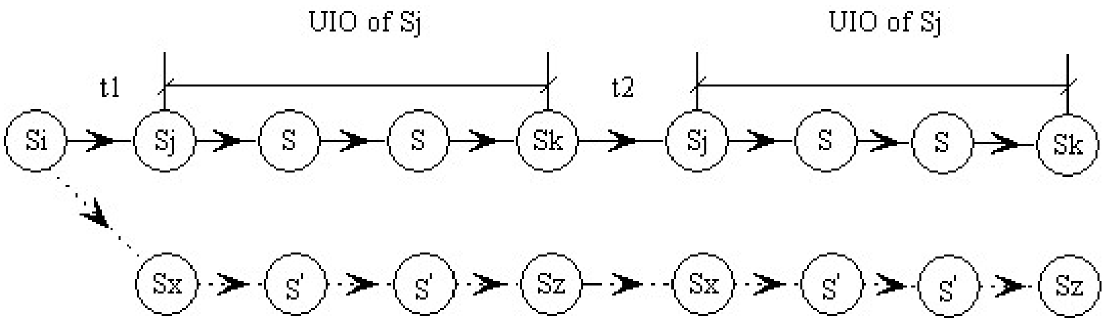


Figure 7.1: Fault Masked UIO Cycling

An example of fault masked UIO cycling is illustrated in Figure 7.1 where a state transfer error occurs in $t_1(s_i \rightarrow s_j)$ first, leading to an erroneous final state s_x. Due to fault masking, the UIO of s_j fails to find the error, moving the FSM to s_z. Suppose, according to the test order, $t_2(s_k \rightarrow s_j)$ is tested after t_1. When responding to the input, the IUT produces the same output as defined in the specification and arrives at s_x. When applying the UIO of s_j, it again fails to find the fault. The UIO of s_j appears in the test twice, in both cases, failing to exhibit an incorrect final state in the observed input/output sequence.

Proposition 7.3.2 *In a $k-degree$ weak test, if the problem of fault masked UIO cycling does not exist and a symptom is observed at the a^{th} input, then the faulty transition that causes the occurrence of the symptom must be within [(a+1-$(k+1)*(L_{UIO(max)}$+$L_{Link(max)}$+1)), a] of the inputs where $L_{UIO(max)}$ is the max length of UIOs and $L_{Link(max)}$ the max length of linking sequences.*

Proof: Similar to Proposition 7.3.1, proof can be obtained by considering the test structure. Since no problem of fault masked UIO cycling exists in the test, if there are k weak UIOs in the test, the faulty final state of a faulty transition can be detected with the maximum steps of $(k+1)*(L_{UIO(max)}+L_{Linking(max)}+1)$. $\square$

Test can be simplified if UIOC sequences are applied for state verification. When UIOCs are used, no linking sequence is required, namely, $L_{Link(max)}$=0.

7.3.2 Reducing the size of a conflict set using transfer sequences

A: Making a hypothesis

Once a conflict set $S_{conflict}$ is defined, it can be refined. A subset of transitions S_r in $S_{conflict}$ can be removed according to Propositions 7.3.1 and 7.3.2. Figure 7.2 demonstrates a paradigm. Let $L_{UIO(max)} = 2$. Suppose a symptom is observed at the i^{th} input where it executes the transition t_7 ($s_f \to s_g$). The conflict set with the max size is then $S_{conflict} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. If the test is a strong test, the faulty transition must be within [(i-3), i] of the inputs, namely, it must be in the subset of transitions $S_f = \{t_5, t_6, t_7\}$ where $S_{conflict} = S_f \cup S_r$ and $S_r = \{t_1, t_2, t_3, t_4\}$.

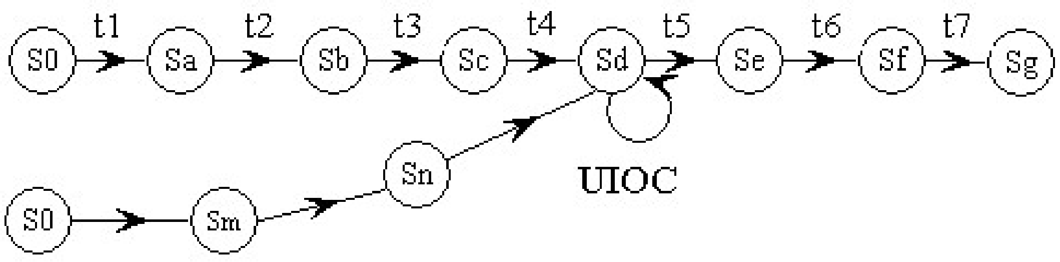


Figure 7.2: Reducing the size of a conflict set by applying transfer sequence

B: Verifying the hypothesis

To verify the hypothesis, a new test sequence is constructed by concatenating a shortest transfer sequence with the inputs that execute t_5, t_6 and t_7 from the original test. The transfer sequence moves the FSM from s_0 to s_d, removing S_r from $S_{conflict}$. In order to increase the confidence that the IUT arrives at an expected final state, the final state is verified by its UIOC sequence.

When the new test sequence is applied to the system, two observations need to be made: 1. have any failures been observed from applying the transfer sequence in the new test sequence? 2. if no failure is exhibited by the transfer sequence, the input/output pairs observed afterwards in the new test sequence need to be compared to those observed after s_d in the original test to check if there exist any differences. If a failure is observed by applying the transfer sequence, transitions executed by the transfer sequence constitutes a new conflict set $S^{'}_{conflict}$ and additional tests need to be developed to isolate the fault. Since the transfer sequence traverses the shortest path from s_0 to s_d, $|S^{'}_{conflict}| \leq |S_{conflict}|$.

Let $tr^{'}_{f}$ be the faulty transition that is identified in $S^{'}_{conflict}$. If $tr^{'}_{f} \in S_{conflict}$, $tr^{'}_{f}$ is defined as the principal faulty transition that causes the occurrence of the observed symptom in the original test. The process of isolating the faulty transition for the observed symptom is then complete. More faults might exist in $S_{conflict}$, these faults can be isolated by constructing some new test sequences where $tr^{'}_{f}$ is not executed or is executed as late as possible; otherwise, if $tr^{'}_{f} \notin S_{conflict}$, one more fault is detected. $tr^{'}_{f}$ needs to be further processed as described in Section 7.4.2. Meanwhile, a new transfer sequence needs to be constructed until no failure is exhibited by this sequence.

Suppose, after applying the transfer sequence, no I/O change is found when the sequence of transitions in S_f is executed, it provides evidence that both the transfer sequence and the input sequence that executes S_r in the original test make the FSM arrive at the same state s_{com}. Since the final state of the transfer sequence is verified by its UIOC sequence, evidence that $s_{com} = s_d$ is provided as well. This further suggests that S_f contains the faulty transition that causes the symptom. Additional tests need to be developed to identify the faulty transition.

Having tested all transitions in $S_{conflict}$, if no faulty transition is defined, it implies that at least one UIO fails to identify the corresponding state. The test is a weak test. Proposition 2 can be applied to estimate the input interval that the faulty transition might fall in. The process starts by considering $[(i+1-(k+1)*(L_{UIO(max)}+L_{Link(max)}+1)), i]|_{k=1}$ first. By removing a set of transitions, if the faulty transition is still not isolated, k is increased by 1. The process repeats until the faulty transition is isolated.

The above considers the situation that no problem of fault masked UIO cycling exists in a test sequence. The existence of such a problem in a test makes the estimation of fault location harder. Fault maskings can be caused either by two different faulty UIOs or a cycled faulty UIO as shown in Figure 7.1. To simplify the estimation, here, a cycled faulty UIO is treated as two or more independent faulty UIOs depending on the number of times this UIO reoccurs. For example, in Figure 7.1, two faulty UIOs are counted for the computation (UIO of s_j appears twice). By such an operation, a k weak test becomes a $k + c$ weak test where c is the sum of times that the cycled faulty UIOs reoccur.

7.3.3 Reducing the size of a conflict set using repeated states

In a conflict set $S_{conflict}$, a state that is the initial state of a transition $tr_a \in S_{conflict}$ may also be the final state of another transition $tr_b \in S_{conflict}$ where tr_b is executed after tr_a. This leads to the repetition of a state when the sequence of transitions in $S_{conflict}$ is executed successively. Transitions between repeated states can be removed to check whether they are responsible for the symptom. Figure 7.3 illustrates the removal scheme.

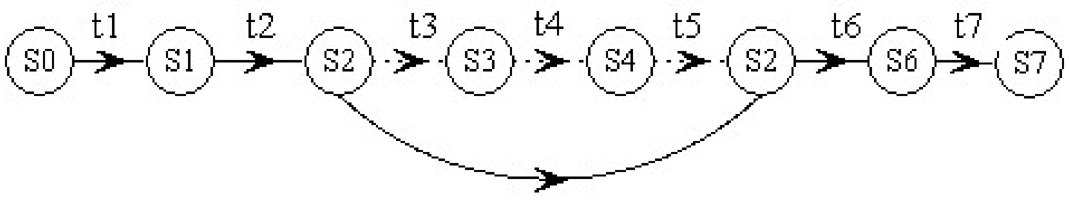


Figure 7.3: Reduce the size of a conflict set by considering the repeated states

In the figure, $S_{conflict}$=$\{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$. It can be noted that s_2 appears twice. A subset of transitions between the repeated state is defined as $S_{cycle} = \{t_3, t_4, t_5\}$. $S_{conflict}$ is then split into two subset $S_{conflict}$=$S_{cycle} \cup S_{remain}$ where $S_{remain} = \{t_1, t_2, t_6, t_7\}$. Based on the original test sequence, a new test sequence is constructed removing the inputs that execute S_{cycle}.

Applying the new test sequence to the system, if, when compared to the original test, the rest of the I/O behaviour remains unchanged, the symptom is then observed in a shorter sequence. The conflict set is consequently reduced to S_{remain}. Additional tests can then be devised to verify the hypothesis.

If, compared to the corresponding I/O segment in the original test sequence, the new test sequence behaves differently, no conclusion can be drawn and, in this situation, the removal scheme of using repeated states does not reduce the size of $S_{conflict}$.

7.4 Identifying a faulty transition

Having reduced the size of a conflict set, further tests need to be devised to identify the fault. Here, the process intends not only to locate the faulty transition, but also to determine its faulty final state.

7.4.1 Isolating the faulty transition

After a conflict set $S_{conflict}$ has been minimised, in order to locate the faulty transition, transitions in $S_{conflict}$ need to be tested individually. Each transition $tr_i \in S_{conflict}$ is tested by moving the FSM to the head state of tr_i, executing tr_i and then verifying tr_i's tail state. In order to increase the reliability, this process should avoid using other untested candidates in $S_{conflict}$.

If, when testing a transition $tr_i \in S_{conflict}$, the use of another untested candidate $tr_j \in S_{conflict}$ is inevitable, one might verify tr_j's tail state as well when it is executed and then apply a transfer sequence to move the IUT back to the tail state of tr_j. If there exists a UIOC sequence for the tail state of tr_j, the UIOC sequence can be applied. Through such an operation, two transitions are tested simultaneously.

The test process described above assumes that UIOs or UIOCs used for state verification are strong UIOs. This, however, might not be true. In order to increase test confidence, one might use a set of test sequences to test a transition $tr_i \in S_{conflict}$, each of which uses a different UIO sequence to verify the final state of tr_i. This, however, requires more test efforts.

7.4.2 Identifying the faulty final state

Once a faulty transition has been located, the faulty final state needs to be identified. This helps to reduce the fault correction effort. A set of estimated erroneous final states $S_{EndState}$ is then constructed.

Let n be the number of states in the FSM. Suppose transition $tr_f : s_i \rightarrow s_j$ is identified as being faulty. It can be noted that, in terms of the alternative final states for tr_f, there are $n-1$ possible mutants. Therefore, $S_{EndState}$ with the max size is $S_{EndState} = \{s_1, ..., s_{j-1}, s_{j+1}, ..., s_n\}$ and $|S_{EndState}| = n - 1$.

The size of $S_{EndState}$ might be reduced by comparing the I/O behaviour exhibited after the faulty transition in the IUT to that defined in the specification.

Definition 7.4.1 *Let $M_S = (I, O, S, \delta, \lambda, s_0)$ be a specification FSM and $M_I = (I, O, S, \delta', \lambda', s_0)$ be an implementation FSM of M_S. Let $x_v \in I$ be an input that executes a transition tr from s_a to s_b in M_I and z_v be the observed output, $z_v = \lambda'(s_a, x_v)$. $S_{VDIS} \subseteq S$ is called the valid defined initial state (VDIS) set of x_v/z_v if $z_v \in O$, $\forall s_v \in S_{VDIS}, \lambda(s_v, x_v) = z_v$ and $\forall s_v \notin S_{VDIS}, \lambda(s_v, x_v) \neq z_v$.*

Let x_v be the input that executes the transition following the isolated faulty transition tr_f in the IUT and z_v be the observed output. The set S_{VDIS} of x_v/z_v is constructed by applying x_v to each state in the specification FSM and comparing the corresponding output with z_v. Let $y_{v(i)}$ be the response from the specification FSM when the machine is in $s_i \in S$ and receives x_v. By comparing $y_{v(i)}$ to z_v, S can be divided into two subsets S_{VDIS} and $\overline{S}_{VDIS}$ where $\forall s_j \in S_{VDIS}$, $\lambda(s_j, x_v) = y_{v_{(j)}} = z_v$ and $\forall s_k \in \overline{S}_{VDIS}$, $\lambda(s_k, x_v) = y_{v_{(k)}} \neq z_v$. If $\overline{S}_{VDIS} \neq \emptyset$, it indicates there exists a non-empty set of states $\overline{S}_{VDIS}$ such that $\forall s_k \in \overline{S}_{VDIS}$, $\lambda(s_k, x_v) \neq z_v$, which suggests that the erroneous final state of tr_f is less likely to be in $\overline{S}_{VDIS}$. $S_{EndState}$ is then reduced to S_{VDIS}.

The size of the estimated faulty final state set might be further reduced by using a set of faulty final state identification test sequences.

Definition 7.4.2 *Let $I = \{a_1, ..., a_k\}$ be the input set of a specification FSM M_S and M_I be an IUT of M_S. Let an isolated faulty transition tr_f of M_I be executed by a test sequence $tv = x_1, ..., x_v$ at the v^{th} input x_v. $TS = \{ts_1, ..., ts_k\}$ is called* *the set of faulty final state identification test sequences (FFSITSs) of tr_f where* [1]$ts_l = tv \cdot a_l$*,* $a_l \in I$*.*

For each $ts_j \in TS$, a set S_{VDIS} can be constructed when a_j is applied to the IUT, denoted by S^j_{VDIS}. The final estimated faulty final state set $S_{EndState}$ can then be reduced to $S_{EndState} = S^1_{VDIS} \cap ... \cap S^k_{VDIS}$.

The complexity of faulty final state identification is determined by the number of states in $S_{EndState}$. This is discussed in Section 7.6.2. If the size of $S_{EndState}$ is reduced, the effort involved in identifying the faulty final state is thus reduced.

Once the size of $S_{EndState}$ is reduced, each state $s_i \in S_{EndState}$ needs to be tested to identify the faulty final state. s_i is checked by moving the IUT from s_0 to s_i with a transfer sequence $Seq_{(transfer)}$, and then applying UIO_{s_i} for s_i. In order to increase test confidence, a set of test sequences, $TV = \{tv_1, tv_2, ..., tv_r\}$, can be applied where $tv_i \in TV$ is constructed by concatenating $Seq_{(transfer)}$ with a different UIO sequence for s_i.

7.5 A case study

A case study is designed to evaluate the effectiveness of the proposed method. A reduced, completely specified and strongly connected specification FSM M is defined in Table 7.1 where the machine has five states. The input set is $I = \{a, b, c, d\}$ and output set $O = \{x, y\}$. In order to simplify the analysis, a set of UIOCs (shown in Table 7.2) is used for state verification. For each state, the first UIOC sequence is used for the generation of the test sequence. The rest of the UIOCs are used to verify hypotheses when diagnosing faults. The [2]maximum length of UIOCs, $L_{UIO(max)}$, is 4. In the implementation M', two faults are injected. They are listed in Table 7.3.

Based upon rural Chinese postman algorithm and UIOCs for state verification, a test sequence ts is generated from M. ts is then applied to M' for fault detection.

[1]Notation "$\cdot$" implies the concatenation of two sequences

[2]Here, $L_{UIO(max)}$ refers to the maximum length of UIOCs that are used for the test generation.

No	Transition	No	Transition
t_1	$s_0 \xrightarrow{a/x} s_1$	t_{11}	$s_2 \xrightarrow{c/x} s_3$
t_2	$s_0 \xrightarrow{d/y} s_2$	t_{12}	$s_2 \xrightarrow{b/y} s_4$
t_3	$s_0 \xrightarrow{c/x} s_3$	t_{13}	$s_3 \xrightarrow{a/x} s_0$
t_4	$s_0 \xrightarrow{b/y} s_4$	t_{14}	$s_3 \xrightarrow{b/y} s_1$
t_5	$s_1 \xrightarrow{d/y} s_0$	t_{15}	$s_3 \xrightarrow{c/x} s_2$
t_6	$s_1 \xrightarrow{a/y} s_2$	t_{16}	$s_3 \xrightarrow{d/y} s_4$
t_7	$s_1 \xrightarrow{c/x} s_3$	t_{17}	$s_4 \xrightarrow{b/x} s_0$
t_8	$s_1 \xrightarrow{b/x} s_4$	t_{18}	$s_4 \xrightarrow{d/y} s_1$
t_9	$s_2 \xrightarrow{d/x} s_0$	t_{19}	$s_4 \xrightarrow{c/x} s_2$
t_{10}	$s_2 \xrightarrow{a/y} s_1$	t_{20}	$s_4 \xrightarrow{a/y} s_3$

Table 7.1: Specification finite state machine used for experiments

After ts is applied to M', a symptom is observed at the 17^{th} input where, according to M, $t_8 : s_1 \xrightarrow{b/x} s_4$ should have been executed (shown in Figure 7.4). The sequence of transitions, $\langle t_1, t_8, t_{19}, t_{10}, t_6, t_9, t_1, t_6, t_{11}, t_{14}, t_8, t_{20}, t_{16}, t_{17}, t_2, t_{10}, t_8 \rangle$, executed by the first 17 inputs constitutes the conflict set of the maximum size, this being $S_{conflict} = \{t_1, t_8, t_{19}, t_{10}, t_6, t_9, t_{11}, t_{14}, t_{20}, t_{16}, t_{17}, t_2\}$.

The size of $S_{conflict}$ is then reduced by applying the proposed heuristics. At first it is assumed that ts is a strong test sequence. The removal scheme is then determined by Proposition 7.3.1. As $L_{Link(max)} = 0$ and $L_{UIO(max)} = 4$, according to Proposition 7.3.1, the faulty transition that causes this symptom must be within [13,17] of the inputs. This hypothesis reduces $S_{conflict}$ to $\{t_{16}, t_{17}, t_2, t_{10}, t_8\}$.

To verify the hypothesis, a shortest transfer sequence, c/x, is applied to move M' from s_0 to s_3, removing the inputs in the original test sequence that successively execute $\langle t_1, t_8, t_{19}, t_{10}, t_6, t_9, t_1, t_6, t_{11}, t_{14}, t_8, t_{20} \rangle$. The final state of the transfer sequence s_3 is afterwards verified by its UIOC sequence. In order to increase test confidence, two UIOC sequences $dada/yyyy$ and bba/yxy for s_3 are applied. After applying $cbba$ and $cdada$ to M', $xyxy$ and $xyyyx$ ($xyyyx \neq xyyyy$) are received respectively. These results imply that (1) t_3 is faulty. It is detected by $dada/yyyy$ but masked by bba/yxy; or, (2) t_3 is correctly implemented but

State	UIOC sequence
s_0	dd/yx $daad/yyyx$
s_1	bca/xxy $baca/xyxy$
s_2	daa/xxy $dadd/xxyy$
s_3	bba/yxy $dada/yyyy$
s_4	$bdab/xyyx$ $aaab/yxxx$

Table 7.2: Unique input/output circuit sequences for each state of the finite state machine shown in Table 7.1.

No	Transition	Mutant
t_3	$s_0 \xrightarrow{c/x} s_3$	$s_0 \xrightarrow{c/x} s_0$
t_{17}	$s_4 \xrightarrow{b/x} s_0$	$s_4 \xrightarrow{b/x} s_4$

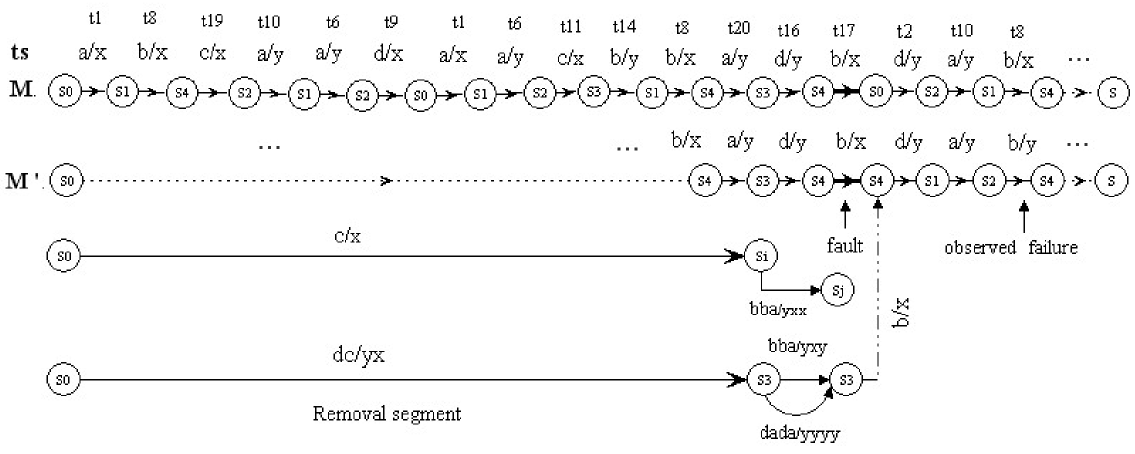
Table 7.3: Injected faults

$dada/yyyy$ traverses a faulty transition, leading to a failure being observed.

To further check the hypothesis, two additional tests $tv_1 = (c/x) \cdot (aaba/xxxy)$ and $tv_2 = (c/x) \cdot (abdba/xyyxy)$ are devised where $aaba/xxxy$ and $abdba/xyyxy$ are two different UIOC sequences for s_3. After applying $caaba$ and $cabdba$ to M', $xxyyy$ and $xxxyxy$ are received, $xxyyy \neq xxxxy$ and $xxxyxy \neq xxyyxy$, which suggests t_3 is faulty and bba/yxy is a weak UIO for s_3.

The erroneous final state of t_3 is further identified as described in Section 7.4. A set of estimated faulty final states for t_3 is constructed by applying a set of faulty final state identification test sequences to M', each test sequence in the set being used to construct the corresponding S_{VDIS}.

Let $S_{VDIS}^{g/h}$ be the S_{VDIS} of g/h where g/h indicates that, when applying g to M', h is observed. After all elements in the input set have been applied, a set of S_{VDIS} can then be obtained. The elements in the set are $S_{VDIS}^{a/x} = \{s_0, s_3\}$, $S_{VDIS}^{b/y} = \{s_0, s_2, s_3\}$, $S_{VDIS}^{c/x} = \{s_0, s_1, s_2, s_3, s_4\}$ and $S_{VDIS}^{d/y} = \{s_0, s_1, s_3, s_4\}$. The



ts
M
M'
t1 t8 t19 t10 t6 t9 t1 t6 t11 t14 t8 t20 t16 t17 t2 t10 t8
a/x b/x c/x a/y a/y d/x a/x a/y c/x b/y b/x a/y d/y b/x d/y a/y b/x
b/x a/y d/y b/x d/y a/y b/y
fault
observed failure
c/x
bba/yxx
b/x
bba/yxy
dc/yx
Removal segment
dada/yyyy

Figure 7.4: Fault detection and identification in $M^{'}$

final estimated faulty final state set is $S_{EndState} = S^{a/x}_{VDIS} \cap S^{b/y}_{VDIS} \cap S^{c/x}_{VDIS} \cap S^{d/y}_{VDIS}$ $= \{s_0, s_3\}$. Additional tests can now be added to verify the hypothesis.

In order to increase test confidence, two test sets $tv_{s_0} = \{(c/x) \cdot (dd/yx),$ $(c/x) \cdot (daad/yyyx)\}$ and $tv_{s_3} = \{(c/x) \cdot (aaba/xxxy),(c/x) \cdot (dada/yyyy)\}$ are devised where s_0 and s_3 are tested respectively. In both tests, two different UIOC sequences are used to verify the corresponding final state. The test results suggest that the erroneous final state of t_3 is s_0.

Since $t_3 \notin S_{conflict}$, a new transfer sequence needs to be constructed to isolate the fault that causes the failure observed in the original test. Still, the $S_{confilict}$ is assumed to be $\{t_{16}, t_{17}, t_7, t_{10}, t_8\}$. Transfer sequence dc/yx is applied, moving $M^{'}$ from s_0 to s_3. In order to increase test confidence, two UIOC sequences, $aaba/xxxy$ and $dada/yyyy$, are used to verify s_3.

After $dcaaba$ and $dcdada$ are applied to $M^{'}$, $yxxxxy$ and $yxyyyy$ are received respectively. This provides evidence that the current state is s_3. Continue to apply those inputs in the original test sequence after the 17^{th} input. By comparing the behaviour to the original test, it is found that the outputs remain unchanged. This increases confidence that the conflict set $S_{conflict} = \{t_{16}, t_{17}, t_7, t_{10}, t_8\}$ contains the faulty transition that cause the observed symptom. Additional tests are required to check each transition in $S_{conflict}$.

When constructing a test sequence, the traversing of t_3 needs to be avoided since it has been found to be faulty.

A set of tests, $VT = \{vt_{t_{16}}, vt_{t_{17}}, vt_{t_2}, vt_{t_{10}}, vt_{t_8}\}$ is devised where $vt_{t_{16}}$, $vt_{t_{17}}$, vt_{t_2}, $vt_{t_{10}}$ and vt_{t_8} check t_{16}, t_{17}, t_2, t_{10} and t_8 respectively. In order to increase test confidence, each test in VT is comprised of two test sequences where two different UIOCs are used to verify the corresponding tail state. These tests are

Transition	Test Set
t_{16}	$vt_{t_{16}} = \{(ac/xx)\cdot(d/y)\cdot(bdab/xyyx), (ac/xx)\cdot(d/y)\cdot(aaab/yxxx)\}$
t_{17}	$vt_{t_{17}} = \{(b/y)\cdot(b/x)\cdot(dd/yx), (b/y)\cdot(b/x)\cdot(daad/yyyx)\}$
t_2	$vt_{t_2} = \{(a/x)\cdot(c/x)\cdot(daa/xxy), (a/x)\cdot(c/x)\cdot(dadd/xxyy)\}$
t_{10}	$vt_{t_{10}} = \{(d/y)\cdot(a/x)\cdot(bca/xxy), (d/y)\cdot(a/x)\cdot(baca/xyxy)\}$
t_8	$vt_{t_8} = \{(a/x)\cdot(b/x)\cdot(bdab/xyyx), (a/x)\cdot(b/x)\cdot(aaab/yxxx)\}$

When applying vt_{t_2} and $vt_{t_{10}}$ to M' no failure is observed, which suggests t_2 and t_{10} are correctly implemented. When applying $vt_{t_{17}}$ to M' both test sequences exhibit a failure which suggests t_{17} is faulty.

When applying $vt_{t_{16}}$ and vt_{t_8} to M', in both tests, one test sequence exhibits a failure while the other shows no error. The test results are $\{(ac/xx)\cdot(d/y)\cdot(bdab/xyyy), (ac/xx)\cdot(d/y)\cdot(aaab/yxxx)\}$ and $\{(a/x)\cdot(b/x)\cdot(bdab/xyyy), (a/x)\cdot(b/x)\cdot(aaab/yxxx)\}$. Through these observations, two hypotheses can be made: (1) t_8 and t_{16} are faulty, and $aaab/yxxx$ is a weak UIO sequence for s_4. A fault is exhibited by $bdab/xyyx$ but masked by $aaab/yxxx$; (2) t_8 and t_{16} are correctly implemented, but $bdab/xyyx$ traverses at least one faulty transition, leading to a failure being observed.

By examining the structure of $bdab/xyyx$, it is found that $bdab/xyyx$ traverses t_{17} that is found to be faulty. It is likely that the second hypothesis is true. To verify the hypothesis, $vt_{t_{16}}$ and vt_{t_8} are replaced with $\{(ac/xx)\cdot(d/y)\cdot(abdb/yyyy), (ac/xx)\cdot(d/y)\cdot(acab/yxyy)\}$ and $\{(a/x)\cdot(b/x)\cdot(abdb/yyyy), (a/x)\cdot(b/x)\cdot(acab/yxyy)\}$. In the tests, $(abdb/yyyy)$ and $(acab/yxyy)$ are two UIOC sequences for s_4 where, according to M, t_{17} is not traversed. After applying *acdabdb*, *acdacab*, *ababdb* and *abacab* to M', *xxyyyyy*, *xxyyxyy*, *xxyyyy* and *xxyxyy* are received respectively. These results suggest that t_8 and t_{16} have been correctly implemented.

The faulty final state of t_{17} is then identified. After all elements in the input set being applied, a set of S_{VDIS} is obtained, this being: $S^{a/y}_{VDIS} = \{s_1, s_2, s_4\}$, $S^{b/x}_{VDIS} = \{s_1, s_4\}$, $S^{c/x}_{VDIS} = \{s_0, s_1, s_2, s_3, s_4\}$ and $S^{d/y}_{VDIS} = \{s_0, s_1, s_3, s_4\}$. The final estimated faulty final state set is $S_{EndState} = S^0_{VDIS} \cap S^1_{VDIS} \cap S^2_{VDIS} \cap S^3_{VDIS} = \{s_1, s_4\}$. Additional tests are then devised to verify the hypothesis.

Two test sequences $ts_1 = (a/x) \cdot (baca/xyxy)$ and $ts_2 = (b/y) \cdot (aaab/yxxx)$ are devised where ts_1 tests s_1 while ts_2 checks s_4. It is concluded that the faulty final state of t_{17} is s_4.

7.6 Complexity

In this section, the complexity of the proposed approach is analysed. The analysis is comprised of two parts - the complexity of fault isolation and the complexity of fault identification. It is shown that the proposed approach can isolate and identify a single fault in low order polynomial time.

7.6.1 Complexity of fault isolation

The complexity of fault isolation is determined by the strength of the UIOs used for the generation of test sequences. The strength of a UIO is its capability to resist fault maskings when required for state verification in the IUT (Nai95). If a symptom is exhibited by a strong test sequence, the conflict set $S_{conflict}$ is of the maximum number $|S_{conflict}|_{max} = L_{UIO(max)} + L_{Linking(max)} + 1$; otherwise, if the test is a k weak test, $|S_{conflict}|_{max} = (k+1) \times (L_{UIO(max)} + L_{Linking(max)} + 1)$, [1]$k \geq 0$. If there exists the problem of faulty masked UIO cycling, the test is treated as a $k + c$ weak test as discussed in the previous sections.

After the conflict set $S_{conflict}$ is constructed, in order to isolate the faulty transitions, each transition $tr_i \in S_{conflict}$ needs to be tested. Let TrS be a set of transfer sequences where $trs_i \in TrS$ is used to move the IUT from s_0 to the initial state of $tr_i \in S_{conflict}$. Let $L_{TrS(max)}$ be the maximum length of the transfer sequences in TrS. The maximum number of steps required for isolating a faulty transition is of $O(|S_{conflict}| \times (L_{TrS(max)} + L_{UIO(max)} + 1))$.

[1]$k = 0$ is equivalent to the case where the test is a strong test.

The process of fault isolation from $S_{conflict}$ considers the use of one UIO sequence for state verification when testing a transition $tr_i \in S_{conflict}$ and assumes this UIO sequence is a strong UIO. However, this might not be true. In order to increase test confidence, a set of test sequences TS_i might be used for the test of a transition tr_i in $S_{conflict}$, each of which uses a different UIO sequence to verify the final state of tr_i.

Let $|TS_i|_{max} = m$, $m \geq 1$. The maximum number of steps required for isolating a faulty transition is then of $O(|S_{conflict}| \times m \times (L_{TrS(max)} + L_{UIO(max)} + 1))$. Therefore, the maximum number of steps required for isolating a single fault is of $O((k+c+1) \times (L_{UIO(max)} + L_{Linking(max)} + 1) \times m \times (L_{TrS(max)} + L_{UIO(max)} + 1))$ where k is the number of faulty UIOs in the test sequence and c is the sum of times that the cycled faulty UIOs reoccur.

7.6.2 Complexity of fault identification

A: Construction of $S_{EndState}$

When identifying the faulty final state of an isolated faulty transition (if the transition holds a state transfer error), a set of estimated faulty final states $S_{EndState}$ needs to be constructed by applying a set of faulty final state identification test sequences. Suppose, in the original test, the faulty transition and the sequence of transitions before this transition are executed by a test segment of length L_{sg}. The number of steps required to construct $S_{EndState}$ is of $O((|I| - 1)(L_{sg} + 1))$ where I is the input set of the FSM. In order that the faulty final state is identified, each state in $S_{EndState}$ needs to be tested.

B: Determining the faulty final state

Let $trs_{shortest}$ with length L_{trs} be the shortest transfer sequence that moves the IUT from s_0 to the initial state of tr_f. Let $|S_{EndState}| = q$, $1 \leq q \leq |S| - 1$. $s_i \in S_{EndState}$ is checked by applying $trs_{shortest}$, executing tr_f with the corresponding input and applying UIO_{s_i}. The length of UIO_{s_i} is less than or equal to $L_{UIO(max)}$. The maximum number of steps required to test $s_i \in S_{EndState}$ is of $O(L_{trs} + L_{UIO(max)} + 1)$. All states in $S_{EndState}$ need to be tested. Therefore, the maximum number of steps required to identify the faulty final state in $S_{EndState}$ is of $O(q \times (L_{trs} + L_{UIO(max)} + 1))$.

Again, the process of faulty final state estimation considers the use of one UIO sequence to verify the corresponding state and assumes this UIO sequence is a strong UIO. In order to increase test confidence, a set of distinct UIOs, $MUIO_i$, may be used to verify state s_i in $S_{EndState}$. Let $|MUIO_i|_{max} = p$, $p \geq 1$. The maximum number of steps required to identify the faulty final state in $S_{EndState}$ is then of $O(q \times p \times (L_{trs} + L_{UIO(max)} + 1))$.

By considering the process of the construction of $S_{EndState}$ together, the maximum number of steps required to identify the faulty final state is of $O((|I| - 1)(L_{sg} + 1) + q \times p \times (L_{trs} + L_{UIO(max)} + 1))$. In the worst case where $q = |S| - 1$, the maximum number of steps is of $O((|I| - 1)(L_{sg} + 1) + p \times (|S| - 1) \times (L_{trs} + L_{UIO(max)} + 1))$, while, in the best case where $q = 1$, the maximum number of steps is of $O((|I| - 1)(L_{sg} + 1) + p \times (L_{trs} + L_{UIO(max)} + 1))$.

7.7 Summary

This chapter investigated fault diagnosis when testing from finite state machines and proposed heuristics to optimise the process of fault isolation and identification. In the proposed approach, a test sequence is first constructed for fault detection. Once a symptom is observed, additional tests are designed to identify the faults that are responsible for the occurrence of the observed symptom.

Based upon the original test, the proposed heuristics are applied to lead to a detected symptom being observed in some shorter test sequences. These shorter test sequences are then used for the construction of a set of diagnosing candidates that is of the minimal size. The minimal set of candidates helps to reduce the cost of fault isolation and identification.

The complexity of the proposed approach was described. A case study was used to demonstrate the application of the approach. In the case study, two state transfer faults were injected into the implementation. These faults were isolated and identified after applying the proposed heuristics.

The case study used in this work considered the use of a comparatively simple example for fault isolation and identification. It is shown how more complicated testing problems such as k degree weak test and fault masked UIO cycling can be catered for. However, more work is required to evaluate these approaches experimentally. This remains a topic for future work.

Chapter 8

Conclusions and future work

Finite State Machines (FSMs) have been considered as powerful means in system modelling and testing. The reviewed literature shows that, once a system is modelled as a finite state machine, it is easy to automate the process of test generation.

This thesis studies the automated generation of test sequences when testing from finite state machines. Three research issues that are highly related to finite state machine based testing were investigated, these being construction of Unique Input/Output (UIO) sequences using Metaheuristic Optmisation Techniques (MOTs), fault coverage in finite state machine based testing, and fault diagnosis when testing from finite state machines.

In the studies of the construction of UIOs, a model is proposed where a fitness function is defined to guide the search for input sequences that are potentially UIOs. In the studies of the improved fault coverage, a new type of Unique Input/Output Circuit (UIOC) sequence is defined. Based upon Rural Chinese Postman Algorithm (RCPA), a new approach is proposed for the construction of more robust test sequences. In the studies of fault diagnosis, heuristics are defined that attempt to lead failures to be observed in some shorter test sequences, which helps to reduce the cost of fault isolation and identification.

The proposed approaches and techniques were evaluated with regard to a set of case studies, which provides experimental evidence for their efficacy.

8.1 Contributions

The declared contributions of this PhD work are summarised as follows:

- proposed a model for the construction of (multiple) UIOs using MOTs (see chapter 5);
- investigated fault coverage when testing from finite state machines and proposed a new method for the generation of more robust test cases (see chapter 6);
- proposed an algorithm for the construction of backward unique input/output sequences (see chapter 6);
- studied fault diagnosis when testing from finite state machines and proposed a set of heuristic rules for fault isolation and identification (see chapter 7).

8.2 Finite state machine based testing

Testing from finite state machines has been discussed in this thesis. It has been demonstrated that finite state machines can be used, not only in the generation of test sequences, but also in the control of the testing process.

Some reviewed work shows how an efficient test sequence can be generated from the finite state machine specification. Based upon rural Chinese postman algorithm, Aho *et al.* (ATLU91) showed that an efficient test sequence may be produced using UIOs for state verification. Shen *et al.* (SLD92) extended the method by using multiple UIOs for each state and showed that this leads to a shorter test sequence. These works, however, do not consider the overlap effect in a test sequence.

Yang *et al.* (YU90) and Miller (MP93) showed that overlap can be used in conjunction with (multiple) UIOs to further reduce the test sequence length. Hierons (Hie96; Hie97) represented overlap by invertible sequences. All of the algorithms guarantee the construction of a test sequence in polynomial time, which makes the finite state machines practical models for testing.

8.3 Construction of UIOs

A very important issue in finite state machine based testing is the construction of Unique Input/Output (UIO) sequences. In finite state machine based testing, the standard test strategy defines that the tail state of a transition needs to be verified once the I/O check is finished. UIOs are often used for state verification. A prerequisite for UIO based testing is that we have at least one UIO sequence for each state of the machine under test. Finding ways to construct UIOs is therefore important.

However, computing UIOs is NP-hard (LY94). Some approaches have been proposed for the construction of UIOs, but they all have some drawbacks. In this work, a model is proposed for the construction of multiple UIOs using Metaheuristic Optimisation Techniques (MOTs), with the sharing techniques. A fitness function, based on properties of a state splitting tree, guides the search for UIOs. A sharing technique is introduced to maintain the diversity in a population by defining a mechanism that measures the similarity of two sequences.

Two finite state machines are used to evaluate the effectiveness of a Genetic Algorithm (GA), GA with sharing, and Simulated Annealing (SA) with sharing. Experimental results show that, when sharing techniques are applied, both GA and SA can find the majority of UIOs from the models under test. This result suggests that it is possible to construct UIOs using MOTs.

8.4 The improved fault coverage

In finite state machine based testing, the problem of fault masking in unique input/output sequences may degrade the test performance of UIO based testing. Two basic types of fault masking are defined in (LS92). Based upon this study, in this work, a new type of Unique Input/Output Circuit (UIOC) sequence is proposed for state verification, which may help to overcome the drawbacks that exist in the UIO based techniques.

UIOCs themselves are particular types of UIOs where the ending states are the same as their initial states. When constructing a UIOC, by further checking the tail state and by using overlap or internal state observation scheme, the abilities of UIOs to resist the problem of fault masking is enhanced. Based upon rural Chinese postman algorithm (RCPA), a new approach for the generation of test sequence from finite state machines is proposed.

The proposed approach was compared with the existing approaches such as F-method and B-method by devising a set of experiments. Experimental results suggest that the proposed approach outperforms or is equal to the existing methods.

It has also been shown that the length of the test sequence generated by using the proposed methods is not always longer than those generated by using the existing methods. In the majority of studies, the test sequences generated by using the proposed method were slightly longer than those generated by using the existing methods. However, in some cases, the proposed method results in some shorter test sequences.

8.5 Fault diagnosis

When testing from finite state machines, a failure observed in the Implementation Under Test (IUT) is called a *symptom*. A symptom could have been caused by an earlier state transfer failure. Transitions that may be used to explain the observed symptoms are called *diagnosing candidates*. Finding strategies to generate an optimal set of diagnosing candidates that could effectively identify faults in the IUT is of great value in reducing the cost of system development and testing.

In this work, we investigated fault diagnosis when testing from finite state machines and propose heuristics for fault isolation and identification. The proposed heuristics attempt to lead a symptom to be observed in some shorter test sequences, which helps to reduce the cost of fault isolation and identification.

A case study was designed to investigate the effectiveness of the proposed method. In the example, two faults were injected to the implementation under test. These faults were identified after applying the proposed heuristics.

8.6 Future work

Three research issues have been investigated. In each case, we have noted some problems. In the studies of the construction of UIOs with MOTs, by applying sharing techniques, a genetic population is forced to form several sub-populations, each of which aims to explore UIOs that are determined as local optima. However, a problem was noted where the distribution of UIOs in a sub-population did not form a good shape. The distribution of the sub-population was dominated by several individuals. In the future work, a new encoding approach might be considered to overcome such a problem.

In the studies of UIOC based testing, overlap and internal state sampling schema were proposed to overcome the problem that a fault is masked by some internal state of a UIO sequence. The study of internal state sampling scheme considered the sampling of one state. The effectiveness of the internal sampling scheme may be further investigated in the future work by considering the use of more than one internal state. Meanwhile, in the experiments, test sequences generated by using the proposed method failed to detect a small number of mutants in some devised finite state machines. The failure of fault detection in UIOC based testing needs to be further studied.

In the studies of fault diagnosis in finite state machine based testing, heuristics were defined that attempt to lead failures to be observed in some shorter test sequences, which helps to reduce the cost of fault isolation and identification. However, the example studied in the work are comparatively simple. Some more complicated testing problems such as k degree weak test and fault masked UIO cycling were only analytically explained but have not been studied by using some examples. These issues need to be investigated in the future work.

References

[AAD04] I. Ahmad, F.M. Ali, and A.S. Das. "LANG - algorithm for constructing unique input/output sequences in finite-state machines". *IEE Proceedings - Computers and Digitital Techniques*, 151:131–140, 2004.

[ABC82] W.R. Adrion, M.A. Branstad, and J.C Cherniavsky. "Validation, Verification, and Testing of Computer Software". *Computing Surveys*, pages 159–192, 1982.

[Abr96] J.R. Abrial. *The B Book - Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AC96] Arlat J. Laprie J-C Avresky, D.R. and Y. Crouzet. "Fault Injection for Formal Testing of Fault Tolerance". *IEEE Transactions on Reliability*, 45(3):443–455, 1996.

[AFE84] A. Ackerman, P. Fowler, and R. Ebenau. "Software inspection and the industrial production of software, Software Validation". *Proceedings of the Symposium on Software Validation*, pages 13–14, 1984.

[AG88] D. Andres and P. Gibbins. *An Introduction to Formal Methods of Software Development.* Milton Keyness, UK: The Open University Press, 1988.

[AHH04] K. Adamopoulos, M. Harman, and R.M. Hierons. "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution". *AAAI Genetic and Evolutionary Computation COnference (GECCO 2004)*, in LNCS 3103:1338–1349, 2004.

[Ake78] S.B. Akers. “Binary decision diagrams”. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[AL91] Crouzet Y. Arlat, J. and J-C Laprie. “Fault injection for the experimental validation of fault tolerance”. *Proceedings in Ann.Esprit Conf. (Esprit’91)*, pages 791–805, 1991.

[And86] S.J. Andriole. *Software Validation, Verification, Testing, and Documentation.* Princeton, NJ: Petrocelli Books, 1986.

[Atk92] A.C. Atkinson. “A segmented algorithm for simulated annealing”. *Statistics and Computing*, (2):221–230, 1992.

[ATLU91] A.V. Aho, A.T. Tahbura, D. Lee, and M.U. Uyar. “An Optimization Technique for Prototol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours”. *IEEE Transactions on Communications*, 39(3):1604–1615, 1991.

[Avr99] D.R. Avresky. “Formal Verification and Testing of Protocol”. *Computer Communications*, 22:681–690, 1999.

[BA82] T.A. Budd and D. Angluin. “Two notations of correctness and their relaton to testing”. *Acta Inf.*, 18:31–45, 1982.

[Bei90] B. Beizer. *Software Testing Techniques.* Thomson Computer Press, 2nd edition, 1990.

[BGM91] G. Bernot, M.-C. Gaudel, and B. Marre. “Software testing based on formal specifications: a theory and a tool”. *IEE/BCS Software Engineering Journal*, 6:387–405, 1991.

[BJG01] J. Bang-Jensen and G. Gutin. *Digraphs: Theory Algorithms and Applications.* Springer-Verlag, London, 2001.

[BM76] J.A. Bondy and U.S.R. Murty. *Graph Theroy with Applications.* Elsevier North Holland, Inc., New York, 1976.

[Boe81] B.W. Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[BP94] G. von Bochmann and A. Petrenko. “Protocol testing: Review of methods and relevance for software testing”. *In Proceedings of the ACM 1994 International Symposium on Software Testing and Analysis*, pages 109–124, 1994.

[BPBM97] G. von Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. “Automating the process of test derivation from SDL specifications”. *in SDL Forum’97*, 1997.

[BPY94] G. von Bochmann, A. Petrenko, and M.Y. Yao. “Fault coverage of tests based on finite state models”. *In IFIP 7th International Workshop on Protocol Test Systems*, pages 91–106, 1994.

[BRM02] M. Brodie, I. Rish, and S. Ma. “Intelligent probing: a cost-efficient approach to fault diagnosis in computer network”. *IBM Systems Journal*, 41(3):372–385, 2002.

[BS83] G. von Bochmann and C.A. Sunshine. “A Survey of Formal Methods”. *Computer Networks and Protocols, P.E.Green, Ed. New York: Plenum*, pages 561–578, 1983.

[BU91] S.C. Boyd and H. Ural. “On the complexity of generating optimal test sequences”. *IEEE Transactions on Software Engineering*, 17:976–978, 1991.

[Bur93] C. J. Burgess. “Software testing using an automatic generator of test data”. *in Proceedings of SQM’93 - Software Quality Management*, pages 541–556, 1993.

[CA92] W. Chun and P.D. Amer. “Improvements on UIO sequence generation and partial UIO sequences”. *Proceedings IFIP WG6.1 12th International Symposium on Protocol Specification, Testing, and Verification*, pages 245–260, 1992.

[CC92] U. Celikkan and R. Cleaveland. “Computing diagnostic tests for incorrect processes”. *Proceedings of IFIP WG6.1 12th International Symposium on Protocol Specification, Testing, and Verification*, pages 263–278, 1992.

[CCK90] M.-S. Chen, Y. Choi, and A. Kershenbaum. “Approaches utilizing segement overlap to minimize test sequences”. *Proceedings of IFIP WG6.1 10th International Symposium on Protocol Specification, Testing, and Verification*, pages 85–98, 1990.

[CCPS98] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatiorial Optimization.* Wiley-Interscience, New York, 1998.

[CDH+03] J. Clark, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, Roper M., and M. Shepperd. “Reformulating Software Engineering as a Search Problem”. *IEE Proceedings - Software*, 150(3):161–175, 2003.

[CHC76] Ramamoorthy CV, S.F. Ho, and W.T. Chen. “On the automated generation of program test data”. *IEEE Transactions on Software Engineering*, 2(4):293–300, 1976.

[Cho78] T.S. Chow. “Testing Software Design Modelled by Finite State Machines”. *IEEE Transactions On Software Engineering*, 4(3):178–187, 1978.

[CHR82] L.A. Clarke, J. Hassell, and D.J. Richardson. “A close look at domain testing”. *IEEE Transactions on Software Engineering*, 8(4):380–390, 1982.

[CK92] A.R. Cavalli and S.U. Kim. “Automated protocol conformance test generation based on formal methods for LOTOS specifications”. *IFIP 5th International Workshop on Protocol Test Systems*, pages 212–220, 1992.

[CK02] M. Chyzy and W. Kosinski. “Evolutionary algorithm for state assignment of finite state machines”. *Proceedings of the Euromicro Symposium on Digital System Design*, pages 359–362, 2002.

[CLBW90] W.H. Chen, C.S. Lu, E.R. Brozovsky, and J.T. Wang. “An optimization technique for protocol conformance testing using multiple UIO sequences”. *Inform. Process. Lett.*, 26:7–11, 1990.

[CM94] J.J. Chilenski and S.P. Miller. “Applicability of Modified Condition/Decision Coverage to Software Testing”. *Software Engineering Journal*, 9(5):193–200, 1994.

[CPR89] L.A. Clarke, A. Podgurski, and D.J. Richardson. “A formal evaluation of data flow path selection critria”. *IEEE Transactions on Software Engineering*, 15(11):1318–1332, 1989.

[CS94] D.A. Carrignton and P.A. Stocks. “A tale of two paradigms: Formal methods and software testing”. *Z User Workshop, Cambridge 1994, Workshops in Computing*, pages 51–68, 1994.

[CSE96] J. Callahan, F. Schneider, and S. Easterbrook. “Automated software testing using model-checking”. *in SPIN’96*, pages 118–127, 1996.

[CTCC98] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. “In black and white: an integrated approach to class-level testing of object-oriented programs”. *ACM Transactions on Software Engineering and Methodology*, 7:250–295, 1998.

[CU95] W.H. Chen and H. Ural. “Synchronizable Test Sequences Based on Multiple UIO Sequences”. *IEEE/ACM Transactions on Networking*, 3(2):152–157, 1995.

[CVI89] W.Y.L. Chan, S.T. Vuong, and M.R Ito. “An improved protocol test generation procedure based on UIOs”. *ACM SIGCOMM89*, pages 178–187, 1989.

[CW96] E.M. Clarke and J.M. Wing. "Formal methods: state of the art and future directions". *ACM Computing Surveys*, 28:626–643, 1996.

[CYL01] S.S. Chao, D.L. Yang, and A.C. Liu. "An automated fault diagnosis system using hierarchical reasoning and alarm correlation". *Journal of Network and Systems Management*, 9(2):183–202, 2001.

[CZ93] S.T. Chanson and J. Zhu. "A unified approach to protocol test sequence generation". *Proceedings of INFOCOM*, pages 106–114, 1993.

[DB99] J. Derrick and E. Boiten. "Testing refinements of state-based formal specifications". *Software testing, Verification and Reliability*, 9:27–50, 1999.

[DGM93] P. Dauchy, M.-C. Gaudel, and B. Marre. "Using algebraic specifications in software testing: a case study on the software of an automatic subway". *The Journal of Systems and Software*, 21:229–244, 1993.

[DHHG04] K. Derderian, R.M. Hierons, M. Harman, and Q. Guo. "Input Sequence Generation for Testing of Communicating Finite State Machines (CFSMs) Using Genetic Algorithms". *AAAI Genetic and Evolutionary Computation Conference 2004 (GECCO 2004)*, in LNCS 3103:1429–1430, 2004.

[DHHG05] K. Derderian, R.M. Hierons, M. Harman, and Q. Guo. "Generating feasible input sequences for extended finite state machines (EFSMs) suing Genetic Algorithms". *AAAI Genetic and Evolutionary Computation Conference 2005 (GECCO 2005)*, pages 1081–1082, 2005.

[Dij72] E.W. Dijkstra. "Notes on structured programming". *Structured Programming*, 1972.

[DLS78] R.A. DeMillo, R. Lipton, and F.G. Sayward. "Hints on test data selection: help for the practicing programmer". *IEEE Computer*, 11:34–41, 1978.

[DN84] J.W. Duran and S.C. Ntafos. ‘An evaluation of random testing”. *IEEE Transactions on Software Engineering*, 10(4):438–444, 1984.

[DO93] R.A. DeMillo and A.J. Offut. “Experimental results from an automatic test case generator”. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, 1993.

[Dow93] K.A Dowsland. *Modern Heuristic Techniques for Combinatorial Problems*. McGraw Hill, 1993.

[DSU90] A.T. Dahbura, K. Sabnani, and M.U. Uyar. “Formal methods for generating protocol conformance test sequences”. *Proceedings of IEEE*, 78(8):1317–1325, 1990.

[EFM97] A. Engels, L.M.G. Feijs, and S. Mauw. “Test generation for intelligent networks using model checking”. *TACAS’97*, in LNCS 1217, 1997.

[Ehr89] W.D. Ehrenberger. “Probabilistic techniques for software verification”. *IAEA Technical Committee Meeting on Safety Implications of Computerised Process Control in Nuclear Power Plants*, 1989.

[EJ73] J. Edmonds and E.L. Johnson. “Matchings, Euler Tours, and the Chinese Postman Problem”. *Mathematical Programming*, 5:118–125, 1973.

[EK72] J. Edmonds and R.M. Karp. “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. *Journal of the ACM*, 19(2):248–264, 1972.

[FBK$^+$91] S. Fujiwara, G. von Bochmann, F. Khende, M. Amalou, and A. Ghedamsi. “Test Selection Based on Finite State Models”. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[FFF62] L.R. Fr. Ford and D.R. Fulkerson. *Flows in networks*. Princeton University Press, Princeton, N.J., 1962.

[FK96] R. Ferguson and B. Korel. "The chaining approach for software test data generation". *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[FS96] R. Fletcher and A.S.M. Sajeev. "A framework for testing object-oriented software using formal specifications". *Ada-Europe'96, International Conference on Reliable Software Technologies*, 1996.

[fSI88] International Organization for Standardization (ISO). "Information processing systems - Open systems interconnections - LOTOS - A formal description technique based on the temporal ordering of observational behaviour". *ISO8807*, 1988.

[fSI89] International Organization for Standardization (ISO). "Information processing systems - Open systems interconnections - Estelle - A formal description technique based on an extended state transition model". *ISO9074*, 1989.

[FW88] P.G. Frankl and J.E. Weyuker. "An applicable family of data flow testing criteria". *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[Gau95] M.C. Gaudel. "Testing can be formal too". *In TAPSOFT'95*, pages 82–96, 1995.

[GB92] A. Ghedamsi and G. von Bochmann. "Test Result Analysis and Diagnosis for Finite State Machines". *Proceedings of the 12th International Workshop on Protocol Test Systems*, pages 244–251, 1992.

[GBD93] A. Ghedamsi, G. von Bochmann, and R. Dssouli. "Multiple Fault Diagnosis for Finite State Machines". *Proceedings of IEEE INFOCOM'93*, pages 782–791, 1993.

[GCG90] C.P. Gerrard, D. Coleman, and R.M. Gallimore. "Formal specification and design time testing". *IEEE Transactions on Software Engineering*, 16:1–12, 1990.

[GG75] J.B. Goodenough and S.L. Gerhart. "Towards a theory of test data selection". *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[GH99] A. Gargantini and C. Heitmeyer. "Using model checking to generate tests from requirements specifications". *in ESEC'99*, in LNCS 1687:146–162, 1999.

[GHHD04] Q. Guo, R.M. Hierons, M. Harman, and K. Derderian. "Computing Unique Input/Output Sequences Using Genetic Algorithms". *Formal Approaches to Testing (FATES'03)*, in LNCS 2931:164–177, 2004.

[GHHD05a] Q. Guo, R.M. Hierons, M. Harman, and K. Derderian. "Constructing Multiple Unique Input/Output Sequences Using Metaheuristic Optimisation Techniques". *IEE Proceedings - Software*, 152(3):127–140, 2005.

[GHHD05b] Q. Guo, R.M. Hierons, M. Harman, and K. Derderian. "Improving Test Quality Using Robust Unique Input/Output Circuit Sequences (UIOCs)". *Information and Software Technology*, accepted for publication, 2005.

[GHHD05c] Q Guo, R.M. Hierons, M. Harman, and K. Derderian. "Heuristics for fault diagnosing when testing from finite state machines". *Software testing, verification and reliability*, under review, 2005.

[Gil61] A. Gill. "State-identification experiments in finite automata". *Information and Control*, 4:132–154, 1961.

[Gil62] A. Gill. *Introduction to The Theory of Finite State Machines.* McGraw-Hill, 1962.

[Glo89] F. Glover. "Future paths for integer programming and links to artificial intelligence". *Computers and Operations Research*, 5:533–549, 1989.

[Gol89] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[Gon70] G. Gonenc. “A method for the design of fault detection experiments”. *IEEE Transactions on Computers*, C-19:551–558, 1970.

[GR87] D.E. Goldberg and J. Richardson. “Genetic Algorithms with Sharing for Multimodal Function Optimization”. *In J.J.Grefenstette (Ed.) Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49, 1987.

[GW88] J.A. Goguen and T. Walker. “Introducing OBJ3”. *Computer Science Laboratory SRI International Report SRI-CSL-88-9*, 1988.

[GY98] J. Gross and J. Yellen. *Graph Theory and Its Applications.* The CRC Press, New York, 1998.

[Had01] C.N. Hadjicostis. “Stochastic Testing of Finite State Machines”. *Proceedings of the American Control Conference*, pages 4568–4573, 2001.

[Hal88] P.A.V. Hall. “Towards testing with respect to formal specification”. *Proceedings of the 2nd IEE/BSC Conference on Software Engineering ’88*, pages 159–163, 1988.

[HCJ98] C.M. Huang, M.S. Chiang, and M.Y. Jiang. “UIO: a protocol test sequence generation method using the transition executability analysis (TEA)”. *Computer Communications*, 21:1462–1475, 1998.

[Hen64] F.C. Hennie. “Fault Detecting Experiments for Sequential Circuits”. *Proceedings of the Fifth Annual Switching Theory and Logical Design Symposium*, pages 95–110, 1964.

[Her76] P. Herman. “A data flow analysis approach to program testing”. *Aust. Comput. J.*, 8(3):92–96, 1976.

[Het88] W.C. Hetzel. *The Complete Guide to Software Testing.* Wellesley, 2nd edition, 1988.

[HH91] P.A.V. Hall and R.M. Hierons. "Formal methods and testing". *Tech. Rep. 91/16, Dept. of Computing, the Open University*, 1991.

[Hie93] R.M. Hierons. *Using Formal Specifications to Enhance The Software Testing Process*. Ph.D. Thesis, Brunel University, United Kingdom, 1993.

[Hie96] R.M. Hierons. "Extending Test Sequence Overlap by Invertibility". *The Computer Journal*, 39(4):325–330, 1996.

[Hie97] R.M. Hierons. "Testing From a Finite-State Machine: Extending Invertibility to Sequences". *The Computer Journal*, 40(4):220–230, 1997.

[Hie98] R.M. Hierons. "Minimizing the cost of fault location when testing from a finite state machine". *Computer Cmmunications*, 22(2):120–127, 1998.

[HJ01] M. Harman and B. Jones. "Search-based software engineering". *Information and Software Technology*, 43(14):833–839, 2001.

[Hoa85] A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International,, 1985.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, University of Michigan Press, 1975.

[How76] W.E. Howden. "Reliability of the path analysis testing strategy". *IEEE Transactions on Software Engineering*, SE-2(3):208–215, 1976.

[How82] W.E. Howden. "Weak mutation testing and completeness of test sets". *IEEE Transactions on Software Engineering*, 8(1):208–215, 1982.

[Hsi71] E.P. Hsieh. "Checking experiments for sequential machines". *IEEE Transactions on Computers*, 20:1152–1166, 1971.

[HT90] R. Hamlet and R. Taylor. "Partition testing does not inspire confidence". *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[HU02] R.M. Hierons and H. Ural. "Reduced Length Checking Sequences". *IEEE Transactions on Computers*, 51(9):1111–1117, 2002.

[HU03] R.M. Hierons and H. Ural. " UIO sequence based checking sequences for distributed test architectures". *Information and Software Technology*, 45:793–803, 2003.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Standard 610.12-1990, 1990.

[IT97] ITU-T. *Recommendation Z.500 Framework on Formal Methods in Conformance Testing.* International Telecommunication Union, Geneva, Switzerland, 1997.

[JES98] B.F. Jones, D.E. Eyres, and H.H. Sthamer. "A Strategy for Using Genetic Algorithms to Automate Branch and Fault-based Testing". *The Computer Journal*, 41(2):98–107, 1998.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[JSE96] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. "Automatic Structural Testing Using Genetic Algorithms". *Software Engineering Journal*, 11(5):299–306, 1996.

[KGJV83] S. Kirkpatrick, C.D. Gelatt, Jr., and M.P. Vecchi. "Optimization by Simulated Annealing". *Science*, 220(4598):671–680, 1983.

[Kho02] A. Khoumsi. "A Temporal Approach for Testing Distributed Systems". *IEEE Transactions on Software Engineering*, 28(11):1085–1103, 2002.

[KK68] I. Kohavi and Z. Kohavi. “Variable-length distinguishing sequences and their application to the design of fault-detection experiments”. *IEEE Transactions on Computers*, C(17):792–795, 1968.

[KMM91] S.M. Kim, R. McNaughton, and R. McCloskey. “A polynomial time algorithm for the local testability problem of deterministic finite automata”. *IEEE Transactions on Computers*, 40:1087–1093, 1991.

[Koh78] Z. Kohavi. *Switching And Finite Automata Theory.* McGraw-Hill, New York, 1978.

[KS95] I. Katzela and M. Schwartz. “Schemes for Fault Identification in Communication Networks”. *IEEE/ACM Transactions on Networking*, 3(6):753–764, 1995.

[Kua62] M.-K. Kuan. “Graphic programming using odd or even points”. *Chinese Math*, 1:273–277, 1962.

[LBP94a] G. Luo, G. von Bochmann, and A. Petrenko. “Test selection based on communicating nondeterministic finite-state machines”. *in The 7th IFIP Workshop on Protocol Test Systems*, pages 95–110, 1994.

[LBP94b] G. Luo, G. von Bochmann, and A. Petrenko. “Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method”. *IEEE Transactions of Software Engineering*, 20:149–161, 1994.

[LK83] J Laski and B. Korel. “A data flow oriented program testing strategy”. *IEEE Transactions on Computers*, 9:33–43, 1983.

[Low93] S. Low. “Probabilistic conformance testing of protocols with unobservable transitions”. *Proceedings of the 1st International Conference on Network Protocols*, pages 368–375, 1993.

[LS92] F. Lombardi and Y.-N. Shen. “Evaluation and Improvement of Fault Coverage of Comformance Testing by UIO Sequences”. *IEEE Transactions on Communications*, 40(8):1288–1293, 1992.

[LW02] J.J. Li and W.E. Wong. “Automatic test generation from communicating extended finite state machine (CEFSM)-based models”. *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 181–185, 2002.

[LY94] D. Lee and M. Yannakakis. “Testing Finite State Machines: State Identification and Verification”. *IEEE Transactions on Computers*, 43(3):306–320, 1994.

[LY96] D. Lee and M. Yannakakis. “Principles and Methods of Testing Finite State Machines - A Survey”. *Proceedings of IEEE*, 84(8):1090–1122, 1996.

[MA00] B. Marre and A. Arnould. “Test sequences generation from LUSTRE descriptions: GATEL”. *in 15th IEEE International Conference on Automated Software Engineering (ASE 2000), Grenoble*, 2000.

[MC98] I. MacColl and D. Carrington. “Testing MATIS: A case study on spefication-based testing of interactive systems”. *in FAHCI98: Formal Aspects of Human Computer Interaction Workshop*, pages 57–69, 1998.

[McM04] P. McMinn. “Search-based software test data generation : a survey”. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[MG83] P.R. McMullin and J.D. Gannon. “Combining testing with formal specifications: a case study”. *IEEE Transactions on Software Engineering*, 9:328–335, 1983.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MK] D. Marinov and S. Khurshid. “Testera: A novel framework for automated testing of Java programs”. *in Proceedings of IEEE 16th ASE*, page 2001.

[ML90] R.E. Miller and G.M. Lundy. “Testing protocol implementations based on a formal specification”. *Protocol Test Systems III*, pages 289–304, 1990.

[MMN+92] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and J.M Voas. “Estimating the probability of failure when testing reveals no failures”. *IEEE Transactions on Software Engineering*, 18(1):33–43, 1992.

[MMS01] C.C. Michael, G. McGraw, and M.A. Schatz. “Generating Software Test Data by Evolution”. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[MMSL96] E.W. McGookin, D.J. Murray-Smith, and Y. Li. “Segmented Simulated Annealing applied to Sliding Mode Controller Design”. *Proceedings of the 13th World Congress of IFA, San Francisco, USA*, Vol D:333–338, 1996.

[Moo56] E.F. Moore. “Gedanken-experiments on sequential machines”. *Automata Studies*, 34:129–153, 1956.

[MP93] R.E. Miller and S. Paul. “On the Generation of Minimal-Length Conformance Tests for Communication Protocols”. *IEEE/ACM Transactions on Networking*, 1(1):116–129, 1993.

[MP94] R.E. Miller and S. Paul. “Structural Analysis of Protocol Specifications and Generation of Maximal Fault Coverage Conformance Test Sequences”. *IEEE/ACM Transactions on Networking*, 2(5):457–470, 1994.

[MRR+53] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. “Equations of State Calculations by Fast Computing Machines”. *Journal of Chemical and Physics*, 21:1087–1092, 1953.

[Mye79] G.J. Myers. *The art of software testing.* John Wiley & Sons, New York, 1979.

[Nai95] K. Naik. “Fault-tolerant UIO Sequences in Finite State Machines”. *Proceedings of the IFIP WG6.1 TC6 Eight International Workshop on Protocol Test Systems*, pages 201–214, 1995.

[Nai97] K. Naik. “Efficient Computation of Unique Input/Output Sequences in Finite-State Machines”. *IEEE/ACM Transactions on Networking*, 5(4):585–599, 1997.

[NC02] N. Niparnan and P. Chongstitvatana. “An improved genetic algorithm for the inference of finite state machine”. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 189–192, 2002.

[NT81] S. Naito and M. Tsunoyama. “Fault detection for sequential machines by transitions tours”. *Proceedings of IEEE Fault Tolerant Comput. Symp., IEEE Computer Soc. Press*, pages 238–243, 1981.

[Nta84] S.C. Ntafos. “On required element testing”. *IEEE Transactions on Software Engineering*, 10(6):795–803, 1984.

[Nta88] S.C. Ntafos. “A Comparison of Some Structural Testing Strategies”. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988.

[OB89] T.J. Ostrand and M.J. Balcer. “The category-Partition Method for Specifying and Generating Functional Tests”. *Communication of the ACM*, 31(6):667–686, 1989.

[Oul91] M. Ould. “Testing - a challenge to method and tool developers”. *Software Engineering Journal*, 6(2):59–64, 1991.

[PB96] A. Petrenko and G. von Bochmann. “On Fault Coverage of Tests for Finite State Specifications”. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.

[Pet01] A. Petrenko. “Fault Model-Driven Test Derivation from Finite State Models: Annotated Bibliography”. *MOVEP*, in LNCS 2067:196–205, 2001.

[PHP99] R.P. Pargas, M.J. Harrold, and R.R. Peck. “Test-data generation using genetic algorithms”. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[PR97] I. Pomeranz and S.M. Reddy. "Test generation for multiple state-table faults in finite-state machines". *IEEE Transactions on Computers*, 46(7):782–794, 1997.

[PR00] I. Pomeranz and S.M. Reddy. "Functional test generation for full scan circuits". *Proceedings of the conference on Design, Automation and Test in Europe*, pages 396–403, 2000.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, 1995.

[Rop94] M. Roper. *Software Testing.* McGraw-Hill Book Company, London, 1994.

[RU95] A. Rezaki and H. Ural. "Construction of Checking Sequences Based On Characterization Sets". *Computer Communications*, 18:911–920, 1995.

[RW85] S. Rapps and E. Weyuker. "Selecting software test data using data flow information". *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[Sad97] S. Sadeghipour. "Test case generation on the basis of formal specifications". *Workshop on Formal Design of Safety Critical Embedded Systems, Munich, Gemany*, 1997.

[SGL01] H. Sun, M. Gao, and A. Liang. "Study on UIO sequence generation for sequential machine's functional test". *Proceedings of the 4th International Conference on ASIC*, pages 628–632, 2001.

[SL89] D.P. Sidhu and T.K. Leung. "Formal Methods for Protocol Testing: A Detailed Study". *IEEE Transactions on Software Engineering*, 15(4):413–426, 1989.

[SL92] X.J. Shen and G.G. Li. "A new protocol conformance test generation method and experimental results". *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's*, pages 75–84, 1992.

[SLD92] Y.N. Shen, F. Lombardi, and A.T. Dahbura. “Protocol Conformance Testing Using Multiple UIO Sequences”. *IEEE Transactions on Communications*, 40(8):1282–1287, 1992.

[SMV93] D.P. Sidhu, H. Motteler, and R. Vallurupalli. “On Testing Hierarchies for Protocols”. *IEEE/ACM Transactions on Networking*, 1(5):590–599, 1993.

[Spi88] J.M. Spivey. *Understanding Z: A Specification languages and its formal semantics.* Cambridge University Press, 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[SS04] M. Steinder and A.S. Sethi. “Probabilistic Fault Diagnosis in Communication Systems Through Incremental Hypothesis Updating”. *Computer Networks*, 45:537–562, 2004.

[SSLS91] X. Sun, Y.-N. Shen, F. Lombardi, and D. Sciuto. “Prototocl conformance testing by discriminating UIO sequences”. *Proceedings of IFIP WG6.1 11th International Symposium on Protocol Specification, Testing, and Verification*, (349-364), 1991.

[SST91] X.J. Shen, S. Scoggins, and A. Tang. “An Improved RCP-method for Protocol Test Generation Using Backward UIO sequences”. *Proceedings of ACM Symposium on Applied Computing (SAC 1991)*, pages 284–293, 1991.

[SVJ98] D.C. Sun, B. Vinnakota, and W.L. Jiang. “Fast State Verification”. *35th Design Automation Conference*, pages 619–624, 1998.

[TCM98] N. Tracey, J. Clark, and K. Mander. “Automated Program Flaw Finding using Simulated Annealing”. *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis*, pages 83–81, 1998.

[TCMM00] N. Tracey, J. Clark, K. Mander, and J. McDermid. “Automated Test-data Generation for Exception Conditions”. *Software Practice and Experience*, 30(1):61–79, 2000.

[Tha94] G. Thaller. *Software Tests for Students and Practisers.* Somepublisher, Wiesbaden, 1994.

[TPB95] Q.M. Tan, A. Petrenko, and G. von Bochmann. “Modelling basic LOTOS by FSMs for conformance testing”. *in IFIP Protocol Specification, Testing, and Verification XV*, pages 137–152, 1995.

[TPB97] Q.M. Tan, A. Petrenko, and G. von Bochmann. “Checking experiments with labeled transition systems for trace equivalence”. *in The 10th International Workshop on Testing of Communicating Systems*, pages 167–182, 1997.

[Tre96] J. Tretmans. “Conformance testing with labelled transition systems:Implementation relations and test generation”. *Computer Networks and ISDN Systems*, 29:49–79, 1996.

[TY98] K.C. Tai and Y.C. Young. “Synchronizable test sequences of finite state machines”. *Computer Networks and ISDN Systems*, 30:1111–1134, 1998.

[UD86] M.U. Uyar and A.T. Dahbura. “Optimal test sequence generation for protocols: the Chinese postman algorithm applied to Q.931”. *in Proceedings of IEEE Global Telecommunications Conference*, pages 68–72, 1986.

[UWZ97] H. Ural, X.L. Wu, and F. Zhang. “On Minimizing the Lengths of Checking Sequences”. *IEEE Transactions on Computers*, 46(1):93–99, 1997.

[VCI89] S.T. Vuong, W.W.L. Chan, and M.R. Ito. “The UIOv-method for protocol test sequence generation”. *in The 2nd International Workshop on Protocol Test Systems*, 1989.

[vdBKP92] S.P. van de Burgt, J. Kroon, and A.M. Peeters. “Testability of formal specifications”. *Proceedings of IFIP WG6.1 12th International Symposium on Protocol Specification, Testing, and Verification*, pages 177–188, 1992.

[WC80] L.J. White and E.I. Cohen. “A domain strategy for computer program testing”. *IEEE Transactions on Software Engineering*, 6(3):247–257, 1980.

[WH88] 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, 1988.

[WHH80] M.R. Woodward, D. Hedley, and M.A. Hennel. “Experience with path analysis and testing of programs”. *IEEE Transactions on Software Engineering*, 6(5):278–286, 1980.

[Whi94] D. Whitley. “A genetic algorithm tutorial”. *Statistics and Computing*, 4:65–85, 1994.

[WO80] E.J. Weyuker and T.J. Ostrand. “Theories of program testing and the application of revealing subdomains”. *IEEE Transactions on Software Engineering*, 6(3):236–246, 1980.

[WS93] C. Wang and M. Schwartz. “Fault Detection With Multiple Observers”. *IEEE/ACM Transactions on Networking*, 1(1):48–55, 1993.

[WSJE97] J. Wegener, H. Sthamer, B.F. Jones, and D.E. Eyres. “Testing Real-time Systems Using Genetic Algorithms”. *Software Quality*, 6(2):127–135, 1997.

[YPB93] M. Yao, A. Petrenko, and G. von Bochmann. “Conformance testing of protocol machines without reset”. *in Protocol Specification, Testing and Verification, XIII(C-16)*, pages 241–256, 1993.

[YU90] B. Yang and H. Ural. “Protocol Conformance Test Generation Using Multiple UIO Sequences with Overlapping”. *ACM SIGCOMM 90: Communications, Architectures, and Protocols*, pages 118–125, 1990.

[Zei83] S.J. Zeil. “Testing for perturbations of program statements”. *IEEE Transactions on Software Engineering*, 9(3):335–346, 1983.

[ZHM97] H. Zhu, P.A.V. Hall, and J.H.R. May. “Software unit test coverage and adequacy”. *ACM Computing Surveys*, 29(4):366–427, 1997.