Automated test sequence generation for Finite State Machines using Genetic Algorithms

A thesis submitted for the degree of
Doctor of Philosophy

Karnig Agop Derderian

School of Information Systems, Computing and Mathematics
Brunel University
Uxbridge, Middlesex
UB8 3PH
United Kingdom

29 September 2006

To my parents.

Table of Contents

List of Figures

Acknowledgements

I would like to thank Professor Rob M. Hierons, my first supervisor, and Professor Mark Harman, my second supervisor, for their help, suggestions and constant support during this research.

I would also like to thank the Department of Information Systems and Computing in Brunel University for the exceptional support that they provide to research students.

Related publications

- Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Input sequence generation for testing of communicating finite state machines CFSMs. *In LNCS vol. 3103* - GECCO 04: Proceedings of the 2004 conference on Genetic and evolutionary computation, pages 1429-1430. Springer, 2004.
- Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms. In *GECCO 05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1081-1082, New York, NY, USA, 2005. ACM Press.
- Karnig Derderian, Robert M. Hierons, Mark Harman, and Qiang Guo. Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs. *The Computer Journal*, 49(3) : 331-344, 2006.

Abstract

Testing software implementations, formally specified using finite state automata (FSA) has been of interest. Such systems include communication protocols and control sections of safety critical systems. There is extensive literature regarding how to formally validate an FSM based specification, but testing that an implementation conforms to the specification is still an open problem.

Two aspects of FSA based testing, both NP-hard problems, are discussed in this thesis and then combined. These are the generation of state verification sequences (UIOs) and the generation of sequences of conditional transitions that are easy to trigger.

In order to facilitate test sequence generation a novel representation of the transition conditions and a number of fitness function algorithms are defined. An empirical study of the effectiveness on real FSA based systems and example FSAs provides some interesting positive results. The use of genetic algorithms (GAs) makes these problems scalable for large FSAs. The experiments used a software tool that was developed in Java (14000 lines with comments).

Chapter 1

Introduction

1.1 About Testing

As computer technology advances, systems are getting larger and fulfil more tasks. As a result the role of testing has become increasingly important as part of the system design and implementation.

Software testing is an important but expensive process. Studies show that it can form about fifty percent of the total development cost [Beiz 90]. It is time consuming and error prone. Different testing techniques exist, but they all aim to increase the quality and confidence of the end product.The two issues in testing that need to be considered are test effectiveness and test efficiency. Increasing test effectiveness could be accomplished by using test cases that have a high likelihood of discovering a fault and increasing test efficiency could be achieved by reducing the number of test cases that need to be executed without significantly compromising effectiveness.

Since one of the costliest aspects of testing is the human effort it would be logical to improve the process by full or partial automation. Hence not only testing could be cheaper and more efficient, but previously intractable problems could be tackled. A lot of work has been done on software test automation and specifically automating test data generation [Sabn 88, Shen 89, Jone 98, Jone 96, Lee 96, Li 94, Petr 04, Dual 04].

This chapter starts by outlining the use of software testing to increase the quality and confidence in a software product. Formal methods that can capture the specification of software unambiguously are discussed next. Then we discuss test generation and how the problems addressed in this thesis relate to the software testing field. Finally the structure of this thesis is briefly outlined.

1.2 Testing as part of the system development process

Validation and verification are essential to the system life cycle. Validation is defined by [Boeh 81] as *Are we building the right system?*, while verification is defined as *Are we building the system right?*. Validation is necessary to ensure a level of confidence that the specification addresses the customer's requirements and verification checks whether the implementation adheres (conforms) to that specification.

Software development has changed from a process of small tasks involving a few people to enormous tasks involving many dedicated teams. This has evolved testing from a small task performed by the software engineer himself to a process present in every stage of the software development life cycle. Hence planning for validation and verification throughout the life cycle has become a necessity. Validation and verification are not mutually exclusive and are highly related to software quality.

1.2.1 Static verification

Static verification involves formal or informal analyses of the specifications, design or the source code of a system. When a formal specification language is used formal analyses can be applied to the specification, design and coding stages. This could simply involve checking that certain essential properties hold, or could involve a formal or informal proof of conformance. If a formal specification language is not used it is not possible to prove conformance to a specification, since it will be ambiguous (described using text and diagrams), however it is still possible to analyse the design and source code.

Informal proofs use step-by-step reasoning to inspect a system, while formal proofs use mathematical logic based on set axioms and inference rules. Formal proofs can be checked automatically and an automatic theorem prover can be used to produce such proofs. While informal proofs can be shorter and simpler for humans to produce, formal proofs can be automatically generated or checked and hence could be less likely to have human introduced errors.

However it can be difficult to prove conformance to the specification for large systems. A process known as refinement can be used to alleviate this. A formal specification can be converted into an implementation using a series of simple refinements, each of which can be proven separately [Derr 99].

1.2.2 Dynamic verification

Dynamic verification involves execution of the system implementation. A number of inputs are chosen and by observing the corresponding outputs the gap between the test model and the implementation is determined. This process is known as *testing*, and the inputs are referred to as *test cases*.

Testing usually starts with *unit* or *module testing*, in which modules are tested individually. In *integration testing* modules are grouped together according to their functionality and tested as integrated subsystems. After this the *system testing* phase tests the system as a whole.

Dynamic verification can be further divided into three categories - black-box testing, white-box testing, and random testing.

Back-box testing

In back-box testing a system is tested against its requirements without having internal knowledge of how the system was implemented. Test cases are generated from the system specification. Only information about what inputs does the system expect and what are the specified outputs is available, without knowledge of how the system derives those results.

Since this no knowledge of the implementation of the system is required in black-box testing, test cases can be designed as soon as the system specifications are complete. They test not only individual system components but also the interaction between them. The test cases are implementation independent.

White-box testing

White-box testing uses information from the internal structure of a system to devise tests to check the operation of individual components or the system as a whole. Black-box and white-box testing both choose test cases that investigate a particular characteristic of the system, however in white-box testing test cases can be generated to test some implementation specific aspects of the system.

Determining the adequacy of test cases is just as important as generating them. There are many types of coverage criteria that guide the effort of test case generation. Three examples of test coverage criteria are *statement coverage*, *branch coverage* and *path coverage*. Statement coverage is achieved when a test set causes every reachable statement of the code to be executed at least once. Branch coverage is achieved when a test set causes every feasible branch of the code to be executed at least once and path coverage is achieved when every feasible execution path of the implementation is taken.

Random testing

In random testing test cases are chosen randomly from the test domain. When only a small fraction of the test domain can effectively be explored some techniques can be used to help. Operational profiles can be used to narrow the search domain and to ensure some uniformity of the test cases. Operational profiles are quantitative characterisation of how a system will be used.

Exhaustive testing where the test set consists of all possible test cases is a special type of random testing. Although exhaustive testing guarantees complete fault coverage, in practice it is usually impossible to accomplish [Andr 86].

1.3 Formal specification languages

Requirements can be captured using formal specification languages or informal specification languages. Formal specification languages have a mathematical basis and have a formal notation to model the system requirements [Andr 88]. Informal specification languages on the other hand use a combination of semiformal textual grammars, free text and graphical representations to represent the system requirements. The clarity of such specifications captured using informal specification language can be affected by factors such as experience of the developer, work or cultural background. Formal specification languages can be used to remove the ambiguity usually associated with informal specification languages.

Capturing the requirements of a system in formal specification languages can be a difficult and expensive task for very large systems. Hence it is usually safety critical systems, that require very high levels of reliability, that are specified using formal specifications.

There are three main types of formal specification languages, these being:

1. Model oriented languages
2. Algebraic specification languages
3. Process algebras

1.3.1 Model oriented languages

Model oriented languages usually model the system by defining states of the system with a number of operators over each states. An operation is a function that maps a value of the state together with values of parameters to the operation onto a new state value by defining the relationship between start (start state and input) and end (end state and output).

The most widely known model oriented specification languages are VDM-SL, the specification language associated with VDM [Jone 90], the Z specification language [Spiv 88, Spiv 89] and the B specification language [Abri 96].

1.3.2 Algebraic specification languages

Algebraic specification languages use methods derived from abstract algebra to capture system requirements. They describe the behaviour of the system in terms of axioms. These axioms are usually formulated as equations each of which can be qualified by a condition. Examples of algebraic specification languages include OBJ3 [Gogu 88] and the Common Algebraic Specification Language (CASL) [Moss 04].

An important advantage of algebraic specification languages is that sections of the system can be easily isolated and simulated for testing using a process known as term rewriting. It involves re-writing the axioms (operations) [Berg 89] in such a way that only part of the system's behaviour is described and hence easier to generate tests for.

1.3.3 Process algebras

Process algebras are often used to describe systems with high degree of concurrency or non-determinism. Process algebras are frequently used to specify the control sections of a protocol, communicating systems and concurrent systems. The best known process algebra languages are CSP [Hoar 85], CCS [Miln 89] and LOTOS [Stan 88].

Process algebras are good at modelling situations where a number of entities need to interact by providing means of defining a set of agents and the way they communicate. These agents can perform internal actions as well as communicate with other agents. Recursion is often used to express infinite behaviour. The ability to express concurrency using process algebras is useful in analysing such specifications in order to check that they cannot lead to deadlock. A system has a deadlock if the system has a set of processes, each of which is blocked, waiting for requirements that can never be satisfied [Coff 71].

Finite state machines (FSMs) [Koha 78] are a less general type of process algebra, but they are well understood since they have been studied for many years. A disadvantage of finite state machines is that they are unable to express communication and nondeterminism as well as some other process algebras. However FSMs have a testing fault model, which allows a variety of test generation algorithms to be used. Chapters 2 and 3 discuss testing of finite state machines.

Other finite state based languages like Statecharts [Hare 98], SDL [Beli 89] and X-machines [Halc 98] allow the representation of additional internal data and are called extended finite state machines [Holz 90]. Sets of communicating finite state machine [Bran 83] are commonly used to model communicating protocols and nondeterminism can be modelled using nondeterministic finite state machine [Hopc 79]. Testing of extended finite state machines is discussed in Chapters 2, 4 and 5.

1.4 Test generation

To ensure (with a certain level of confidence) that an implementation conforms to its specification sufficient testing has to be done. Two factors, *test cost* and *fault coverage* are tightly related when evaluating a test case. Test cost usually reflects the amount of test data that has to be used to test the implementation while fault coverage reflects the number of potential faults in the implementation that can be detected with a given test case. It is always desirable to maximise fault coverage and minimise test costs. Since only exhaustive testing can guarantee that an implementation fully reflects the functionality of its specification, a testing strategy has to be selected that provides a compromise between fault coverage and test costs.

In terms of observing the internal behaviour of a system implementation, testing can be categorised as black-box or white-box. In black-box testing the tester does not have information on the structure of an implementation and constructs test cases directly from the specification. The test cases are generated from the specification and the implementation is tested by observing its output. Test design can start immediately after the specification is complete and faults introduced during the implementation stage may be easier to find. However test cases with complete fault coverage may be difficult to construct without observing the internal behaviour of the implementation. White-box testing accesses the internal structure of the implementation and observes information about the execution step by step, which allows more rigorous analysis of what a test case achieves. However it is sometimes difficult to examine all the implementation code and this process often requires a very skilled tester.

Control flow testing is a white-box testing approach that uses the knowledge of the control structure of the program under test. The control structure of the program can be represented by a control flow graph where a syntactic unit like a predicate in a branch is represented by a node and edges represent the nodes that are reachable through execution. A variety of test coverage criteria can be used like statement coverage [Ntaf 88, Beiz 90], branch coverage [Ntaf 88], condition coverage [Beiz 90, Chil 94] or path coverage. Each coverage criterion has its strengths and weaknesses and is usually not used in isolation.

Data flow testing [Rapp 85] focuses on data dependencies within a program, how values are associated with variables. Data flow testing aims to test some subset or some subset of set of combination of these dependencies.

Partition analysis [Ostr 88] is a method that derives the input domain in subdomains which should have equivalent behaviour according to the specification. Faults can either lead to incorrect output (computational faults) or incorrect partitions (domain faults). Only a few inputs from each subdomain can be sufficient to produce test cases with high fault coverage.

Mutation testing [DeMi 78] aims to distinguish a program from a set of deliberately faulty versions of the program (mutants). It introduces faults within the program and measures the ability of test cases to distinguish between the original program and those mutants. A particular problem for mutation testing is identifying equivalent mutants and generating test cases for mutants that are hard to distinguish from the original program.

Random testing is another testing method. Using operational profiles can help guide test case generation using random testing. However it is often difficult to produce a complete operational profile for a system, hence test case generation using operational profile might not always provide the required confidence in the system.

One of the big problems with all testing is knowing when to stop. Also it is hard to determine if the whole of the system has been tested and that there are no faults. Statistical methods can be used to estimate the likelihood of undetected faults remaining in a system under test. Such methods can be used to direct the scope of test case generation. Due to the scale of systems toady it is often difficult to prove a fault free implementation. However a level of confidence in the reliability of a system can be estimated using some test case generation techniques. One advantage for using state based testing, and a motivation for this research, is the existence of statistical methods that guide the test case generation.

1.4.1 Search-based testing

Search based techniques like genetic algorithms (GAs) have recently been applied to the generation of test cases. The problem of generating test cases is transformed to a search problem with a vast search space. Traditional search algorithms would be too costly to apply. Some problems in testing are NP-hard and metaheuristic search based techniques have been proved to be efficient in providing good solutions to such problems.

Test case generation using search based techniques is relatively simple. A method for representing the test cases in the search space has to be defined, which determines the size and shape of the search space. A method for estimating the cost or effectiveness of test sets according to a particular criterion is also defined and used to guide such algorithms in the search for test sets. One of the main challenges is defining such estimation methods that are effective and efficient.

GAs are an evolutionary search based technique based on the Darwinian logic of evolution. Although the efficiency of a search technique is related to the shape of the search space, GAs have been shown to perform well in a variety of unknown domains. This could explain the popularity of GAs in search based research even beyond software testing and computer science. GAs are discussed in Chapter 2.

GAs have been of interest in structural coverage testing [Jone 98], mutation testing [Bott 02], exception detection [Trac 00] and worst case and best case execution time [Wegn 97]. A review of search based test data generation can be found in [McMi 04].

However in search based testing it is sometimes difficult to define the termination criterion, a problem associated with searching unknown search spaces. Exploring the entire search space is similar to exhaustive testing. Currently this problem is addressed by specifying the execution time or execution cycles.

Chapters 3, 4 and 5 present work on using GAs to generate test cases for finite state machines and extended finite state machines.

1.4.2 Testing from finite state machines

There has been much interest in the use of formal specifications because formal specifications, being mathematical structures, can be used to automate test case generation.

Finite state machines have been used to model systems in different areas like sequential circuits [Henn 64], software development [Chow 78] and communication protocols [Aho 91, Lee 94, Sidh 89, Shen 89, Yang 90, Mill 93, Shen 92, Tane 96]. When testing against a finite state machine we can only test that the specification has been correctly implemented. This is usually done by observing the output behaviour of the implementation and comparing that to the specified output behaviour. This process is known as conformance testing and it is discussed in Chapter 2. Generating test cases for finite state machines and extended finite state machines using GAs is discussed in Chapters 3, 4 and 5.

1.5 The structure of this thesis

This thesis is divided in four connected sections. The next section (Chapter 2) further clarifies the context of this work by introducing finite state machines and how they can be tested for conformance. Chapter 2 also introduces some heuristic search techniques, in particular genetic algorithms.

The automated generation of unique inpout/output sequences (UIOs) for finite state machines using genetic algorithms is presented in Chapter 3. Some of work in this chapter was published in a conference proceedings [Derd 05] and the findings were presented in a journal publication [Derd 06].

Chapter 4 evaluates how to estimate the ease of execution of transition paths in extended finite state machines. A search for such transition paths using genetic algorithms is evaluated.

Chapter 5 examines how the work in Chapters 3 and 4 can be combined to generate transition paths in extended finite state machines that have state verification properties and are easy to trigger. Genetic algorithms are used again as the search heuristic.

Conclusions are drawn in Chapter 6. The relevance of these results in the context of communicating finite state machines is discussed and the topics of future work are outlined. Some of the work on communicating finite state machines was published in a conference proceedings [Derd 04].

This thesis addresses problems associated with automating test case generation for systems specified using finite state machines. It attempts to define easy to compute test case adequacy estimations and use GAs in attempt to generate test cases with specific properties. A set of algorithms, data abstractions and a software tool in Java were developed to assist the experimental approach to validate the hypothesis. The experiments compared test cases generated by the suggested test case adequacy estimations using GAs and compared results with randomly generated test cases produced with equivalent computational effort. A set of real systems and a set of larger, randomly generated systems were used. The results showed that in most cases significantly more adequate test cases were generated using GAs. The test case adequacy estimations appeared to have a fair correlation to actual test adequacy estimated using well known techniques in the field. The approach seemed to scale well as larger test cases were considered. Overall the results indicated positive achievements in this work and attention was drawn to some related topics for future work. The work in this thesis can be beneficial when testing large systems that are formally specified using finite state machines, like some systems in the telecommunications industry.

Chapter 2

Preliminaries

2.1 Introduction

Finite state machines (FSMs) have been used to model systems in different areas like sequential circuits [Henn 64], software development [Chow 78] and communication protocols [Aho 91, Lee 94, Sidh 89, Shen 89, Yang 90, Mill 93, Shen 92, Tane 96]. Often such systems are specified using extended finite state machines or (EFSMs) and communicating finite state machines (CFSMs) which can be transformed to FSMs. To ensure the reliability of these systems once implemented they must be tested for conformance to their specification. Usually the implementation of a system specified by an FSM is tested for conformance by applying a sequence of inputs and verifying that the corresponding sequence of outputs is that which is expected.

The chapter begins with some preliminaries on finite state machines and conformance testing. Extended finite state machines are introduced next as are some of the issues that arise when testing them. Next communicating finite state machines are briefly introduced. Finally the genetic algorithms used in this thesis are outlined.

2.2 Finite state machines (FTPs)

Finite state systems are generally modelled using Mealy machines (Moore machines are also used in some domains and equivalence of Moor and Mealy machines is well known [Hopc 79]) that produce an output for every transition triggered by an input. A finite state machine M can be denoted $M = (S, s_1, \delta, \lambda, X, Y)$ where S, X, Y are finite nonempty sets of states, input symbols and output symbols respectively and $s_1 \in S$ is the initial state. δ is the state transition function and λ is the output function. A transition is represented as $t = (s_i, x, y, s_j)$ where $s_i \in S$ is the start state, $s_j \in S$ is the end state, $x \in X$ is the input and $y \in Y$ is the output. When a machine M in state $s_i \in S$ receives input x it moves to state $\delta(s_i, x) = s_j$ and outputs $\lambda(s_i, x) = y$. The functions δ and λ can be extended to take input sequences to give functions $\delta*$ and $\lambda*$ respectively. FSMs can be represented using state transition diagrams where the vertices correspond to states and the edges to transitions which are labelled with the associated input and output [Lee 96] (fig. 2.1).

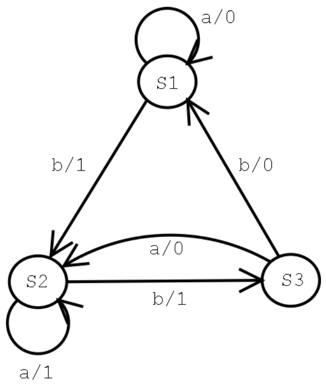


Figure 2.1: Transition diagram of a finite state machine M_0

An FSM is said to be deterministic if there is no pair of transitions that have the same initial state and input i.e. upon an input a unique transition follows to the next state. If for any state an input could trigger more than one transition the machine is nondeterministic. FSMs for which a transition exists for every input $a \in X$ and state $s \in S$ are known as completely (fully) specified. Given an FSM that is partially specified it is possible to apply a completeness assumption and complete M by either adding an error state or assuming that where the input was not specified originally an empty output should be produced.

Those FSMs where every state can be reached from the initial state are known as initially connected. Unreachable states can be removed from any FSM to make it initially connected. An FSM M is strongly connected if for every pair of states (s_i, s_j) from M there is some input sequence that takes M from s_i to s_j. A reset operator takes the FSM to its initial state. The presence of a correctly implemented reset operator is sometimes important for transition testing but cannot always be guaranteed. If M is initially connected and has a reset operator then it must be strongly connected.

Two states s_i and s_j are said to be equivalent if for every input sequence $x`$ we have that $\lambda^*(s_i, x`) = \lambda^*(s_j, x`)$. Otherwise the two states are inequivalent and there exists an input sequence $x`$ where $\lambda^*(s_i, x`) \neq \lambda^*(s_j, x`)$ and that sequence is known as a separating sequence. Comparing FSMs is similar. Two FSMs M and M' are equivalent if for every state in M there is an equivalent state in M' and vice versa. A minimal FSM is a machine M such that there is no equivalent FSM M' with fewer states than M.

For example Figure 2.1 represents the deterministic FSM M_0. M_0 with an initial state s_1 is initially and strongly connected as every state in M_0 is reachable from any other state. M_0 is also completely specified and minimal.

In this work we consider only deterministic FSMs. For non-deterministic FSM conformance testing refer to [Petr 96, Hwan 01, Hier 04]. It is also safe to assume that only minimal FSMs should be considered as any deterministic FSM can be minimised [Moor 56] and there are well known methods to automatically do so [Koha 78, Moor 56]. Also for the reasons outlined before only strongly connected FSMs are considered.

2.3 Conformance testing

When testing from an FSM model M it is assumed that the implementation under test (IUT) can be modelled by an unknown FSM M' and thus that testing involves comparing the behaviour of two FSMs. Verifying that M' is equivalent to M by only observing the input/output behaviour of M' is known as conformance testing or fault detection.

Often a fault can be categorised as either an output fault or a state transfer fault. Output faults are those faults where the wrong output is produced by a transition and state transfer faults are those faults where the state after a transition is wrong. An output fault can be detected by executing a transition and observing its output. A state transfer fault can be detected by checking if the final state is correct after the transition is executed. Suppose we wish to check a transition $t = (s_i, x, y, s_j)$. The test strategy would involve moving M' to s_i, applying the input x, verifying that the output is y, and using a state verification technique to verify the transition's end state [Chow 78].

The first step is known as homing a machine to a desired initial state s_i. It can be done by using a homing sequence which can be constructed in polynomial time [Koha 78]. The second step, transition verification, is to check whether M' produces a desired output sequence. The last step is to check whether M' is in the expected state $s_j = \delta(s_i, x)$. There are three main techniques that can be used in state verification:

- Distinguishing sequence (DS)
- Unique input/output sequence (UIO)
- Characterizing set (CS)

A distinguishing sequence is an input sequence that produces unique output for each state. Not all FSMs have a DS.

A UIO for state s is an input/output sequence x/y such that $\lambda^*(s, x) = y$ and $\forall s' \in S.s' \neq s$, we have that $\lambda^*(s', x) \neq y$. A DS defines a UIO for every state. While not every FSM has UIOs for all states, some FSMs without a DS have UIOs for all states. Also in practice most FSMs have UIOs for all states [Yang 90].

A characterizing set is a set of input sequences W which can distinguish any pair of states. If every sequence in W is executed from some state s_j, the set of output sequences verifies s_j. However this technique requires a number of input sequences to be executed for each state, and therefore could lead to long test sequences. For some states not every element of W is required and some subset can be used (the Wp method). This can reduce the effort involved in verifying a state. Some improvements to the W-method are presented in [Luo 94b, Petr 96, Luo 94a].

A general method for constructing minimal length checking sequences described in [Inan 99] utilises DSs, characterizing sets or UIOs depending on their existence.

In order to minimise test sequence length when testing using UIOs, usually minimal UIOs are used (the shortest UIO for a state). However it has been suggested [Naik 95] that using non minimal UIOs can improve the chance of avoiding fault masking (when two or more faults collectively mask their faulty behaviour leading to false confidence in the implementation under test). Different UIOs for the same state can be compared by using a metric known as degree of difference (DoD) [Son 98]. The DoD between two transition walks with identical input sequence is defined as the number of output differences between them. A UIO with higher DoD is expected to be more fault tolerant [Naik 95].

Some UIOs could be of exponential length. Generally if a UIO is longer than of $O(n^2)$ it might not be worth considering since a characterizing set with upper bound of $O(n^2)$ length would exist [Chow 78].

Not all FSMs are completely specified. There are two types of conformance testing, strong and weak, depending on how unspecified transitions are treated. In strong conformance testing a completeness assumption stating how missing transitions are to be treated is necessary for partially specified FSMs. In weak conformance testing the missing transitions are treated as 'don't care' and the implementation is required to have only the same 'core behaviour' as the specification.

UIOs have been popular [Sidh 89, Wang 87] since they help in state transition fault detection and tend to yield shorter test sequences than the D and W methods [Sidh 89, Wang 87]. UIOs do not necessarily need a reliable reset operator. The U-method and the T-method have been used for weak conformance testing of partially specified FSMs [Sidh 89]. However the T-method does not check for state transition faults.

In order to test a transition of an FSM the machine has to be put in the initial state of that transition. Then the input is applied and the output checked to verify that it is as expected. After that the UIO sequence for that state is used to verify that there is no state transfer fault. Several test sequence generation techniques based on UIOs can be used [Sidh 89, Aho 91, Shen 89, Yang 90, Shen 91, Shen 92]. This motivates an interest in automating the generation of UIOs.

2.4 Extended finite state machines (EFSMs)

FSMs are known to model appropriately sequential circuits and control portions of communication protocols. However ordinary FSMs are not powerful enough for some applications where EFSMs are used instead. EFSMs have been widely used in the telecommunications field, and are also now being applied to a diverse number of other areas ranging over aircraft, train control, medical and packaging systems. Examples of languages based on EFSMs include SDL, Estelle and Statecharts.

As EFSMs we refer to those Mealy (finite state) machines with parameterised input and output, internal variables, operations and predicates defined over internal variables and input parameters. The form of EFSM used in this thesis is referred to as Normal Form EFSM (NF-EFSMs). It is a general EFSM model that can be obtained directly from SDL specifications [Hier 03].

A Normal Form extended finite state machine (NF-EFSM) M can be defined as $(S, s_1, V, \sigma_0, P, I, O, T)$ where

- S is the finite set of logical states
- $s_1 \in S$ is the initial state
- V is the finite set of internal variables
- σ_0 denotes the mapping from the variables in V to their initial values
- P is the set of input and output parameters
- I is the set of input declarations
- O is the set of output declarations
- T is the finite set of transitions.

A transition $t \in T$ is defined by (s_s, g_I, g_D, op, s_f) where

- s_s is the start state of t;
- g_I is the input guard expressed as (i, P^i, g_{P^i}) where
  - $i \in I \cup NIL$;
  - $P^i \subseteq P$; and
  - g_{P^i} is the input parameter guard that can either be nil or a logical expression in terms of variables in V' and P' where $V' \subseteq V$, $\emptyset \neq P' \subseteq P^i$;

- g_D is the domain guard and can be either nil or represented as a logical expression in terms of variables in V' where $V' \subseteq V$;
- op is the sequential operation which is made of simple output and assignment statements; and
- s_f is the final state of t.

The label of a transition in an NF-EFSM has two guards that decide the feasibility of the transition: the input guard g_I and the domain guard g_D. In order for a transition to be executed g_I, the guard for inputs from the environment must be satisfied. Some inputs may carry values or specific input parameters and M may guard those values with the input parameter guard g_P. The input guard $(NIL, \emptyset, NIL)$ represents no input being required, which makes the transition spontaneous. g_D is the guard, or precondition, on the values of the system variables (e.g. $v > 4$, where $v \in V$). Note that in order to satisfy the domain guard g_D of a transition t, it might be necessary to have taken some specific path to the start state of t. op is a set of sequential statements such as $v := v + 1$ and $!o$ where $v \in V$, $o \in O$ and $!o$ means 'output o to the environment'. Literal outputs (output directly observable by the user) are denoted with ! and output functions (an output functions may produce different output depending on the parameters it receives) without it (e.g. $!o$ and $u(v)$). If a transition t has an output function, then the output value produced by t is determined by the parameters passed to the output function (the parameters could be internal variables or input parameters for that transition). The operation of a transition in an NF-EFSM has only simple statements such as output statements and assignment statements, no branching statements are allowed.

We assume that none of the spontaneous transitions in an NF-EFSM are without any guards, $g_I = (NIL, \emptyset, NIL)$ and $g_D = NIL$, because they will be uncontrollable. When a transition in an NF-EFSM is executed, all the actions of the operation specified in its label are performed consecutively and only once.

Definition 2.4.1. An NF-EFSM is deterministic if for every input sequence x with associated input and domain guard predicate results there is no more than one output sequence that may be produced by the NF-EFSM in response to x.

Upon an input, an NF-EFSM would trigger a given transition depending not only on the input, but on the values of the internal variables. Hence NF-EFSM may be called dynamically deterministic. In FSM there are no internal variables and guards on the transitions. This makes triggering a transition in EFSMs more complex than in FSMs.

Definition 2.4.2. An NF-EFSM is strongly connected if for every ordered pair of states (s, s') there is some feasible path from s to s'.

We assume that any NF-EFSM considered is deterministic and strongly connected.

As example consider the Initiator process of the Inres protocol [Hogr 91] represented as an NF-EFSM on Figure 4.1.

2.5 Testing EFSMs

A system specified by an FSM or an EFSM is tested for conformance by applying a sequence of inputs and verifying that the corresponding sequence of outputs observed is that expected. Typically the machines that arise are complex and brute force exponential testing is infeasible [Lee 96].

In order to minimise manual testing and hence software production costs and speed the process up, automation is necessary. The limitations of manual testing have caused wide use of automation in testing and test data generation [Sabn 88, Shen 89, Jone 98, Jone 96]. Automating the generation of test sequences for EFSMs has been one of the interests [Lee 96, Li 94, Petr 04, Dual 04].

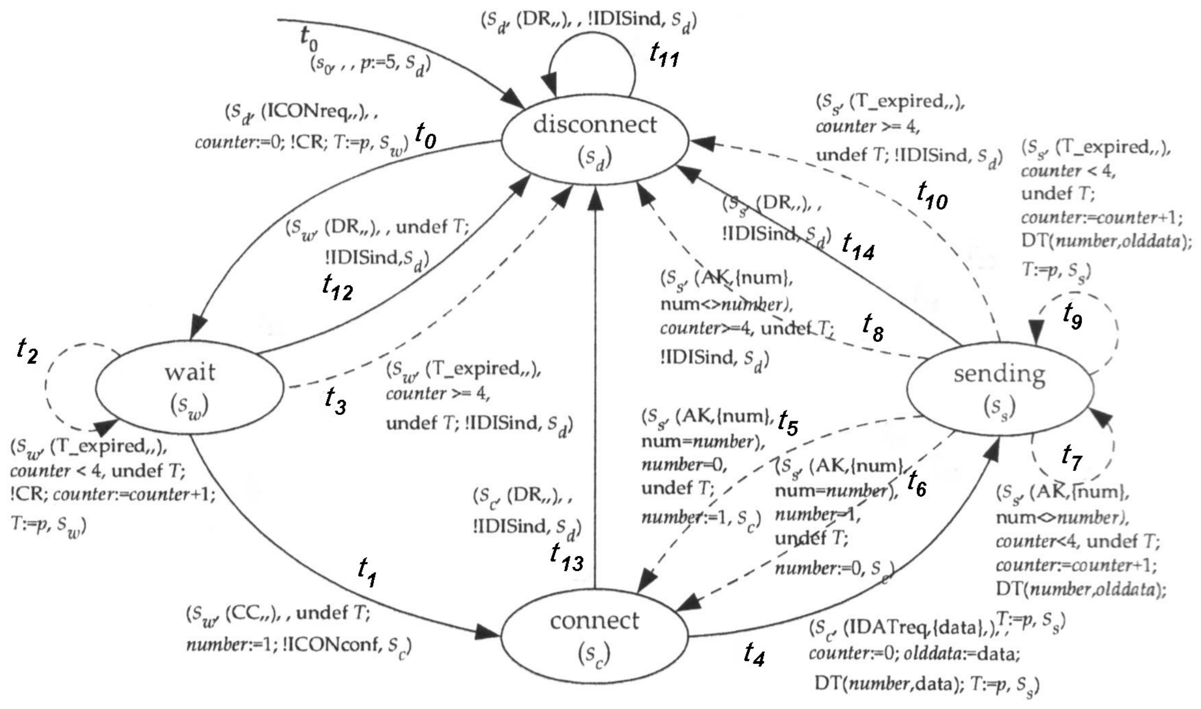

t0
(s0, , p:=5, sd)
(sd, (DR,,), , !IDISind, sd)
t11
(sd, (ICONreq,,), , counter:=0; !CR; T:=p, sw)
t0
disconnect (sd)
(ss, (T_expired,,), counter >= 4, undef T; !IDISind, sd)
t10
(ss, (T_expired,,), counter < 4, undef T; counter:=counter+1; DT(number,olddata); T:=p, ss)
(sw, (DR,,), , undef T; !IDISind, sd)
t12
(ss, (DR,,), , !IDISind, sd)
t14
(ss, (AK,{num}, num<>number), counter>=4, undef T; !IDISind, sd)
t8
t9
t2
wait (sw)
t3
(sw, (T_expired,,), counter >= 4, undef T; !IDISind, sd)
sending (ss)
(sw, (T_expired,,), counter < 4, undef T; !CR; counter:=counter+1; T:=p, sw)
(ss, (AK,{num}, num=number), number=0, undef T; number:=1, sc)
t5
(ss, (AK,{num}, num=number), number=1, undef T; number:=0, sc)
t6
t7
(ss, (AK,{num}, num<>number), counter<4, undef T; counter:=counter+1; DT(number,olddata); T:=p, ss)
(sc, (DR,,), , !IDISind, sd)
t13
t1
(sw, (CC,,), , undef T; number:=1; !ICONconf, sc)
connect (sc)
t4
(sc, (IDATreq,{data},), , counter:=0; olddata:=data; DT(number,data); T:=p, ss)

Figure 2.2: Inres protocol as an EFSM

In EFSMs test sequence generation is more complex than it is for FSMs. In FSMs all of the paths of the directed graph are valid since there are no conditions on the edges and actions do not affect the graph traversal [Dual 00]. With EFSMs however the transition path depends on the result of the input parameter guard and the domain guard. The result of these guards is determined dynamically depending on the values of the internal variables and input declarations, which in turn can assume different values after each transition. Hence some transition paths might have no conditions associated, some might have conditions that are difficult to satisfy and some transition paths will be infeasible. The existence of such infeasible transition paths creates difficulties for automating the test generation process for EFSMs.

One way of approaching the problem is to abstract away the data part of the EFSM and consider it as an FSM on its own. However a transition sequence for the underlying FSM of an EFSM cannot be guaranteed to be feasible for the actual EFSM. Another way is to expand an EFSM to an FSM and then use the techniques used for FSMs. However this can lead to a combinatorial explosion.

Some recent work on generating feasible conformance test sequences for EFSMs was presented in [Dual 04]. It describes how to remove variable interdependencies between the actions and conditions of a certain class of EFSMs, those with linear functions and conditions. The inconsistency removal problem, exponential in complexity for the general software, used there takes advantage of localised inconsistencies to reduce the algorithm complexity and bound it by the size of the subgraphs involved. After the inconsistencies removal, FSM based methods are used for test sequence generation. Our EFSM model is more general as we do not consider only linear constraints. [Dual 04] gives a comprehensive summary of all the important work in this field up to now. Test generation for EFSMs is still an open research problem [Lee 94, Dual 04]

Hence generating a feasible transition path and a corresponding input sequence for EFSMs can help with test sequence generation for EFSMs. The general problem of finding a (an arbitrary) feasible transition sequence for an EFSM and generating the necessary input sequence to trigger, depending on the variables and conditions involved, is generally uncomputable. While a random algorithm could be used it does not always produce acceptable results. In [Guo 04] and [Derd 06] (as well as Chapter 3) GAs are used to generate UIOs for FSMs more efficiently than a random algorithm can. GAs are used to generate easy to trigger transition paths for EFSMs in [Derd 05] (as well as Chapter 4). [Derd 04] (as well as Chapter 6) suggests how GAs can be used to generate input sequences for testing communicating FSMs.

For the remaining of this thesis we will refer to NF-EFSMs simply as EFSMs.

The objective of Chapter 4 is to facilitate the generation of feasible transition paths in EFSMs and the necessary input to trigger them. This can be helpful in test data generation for EFSMs. In Chapter 4 we focus on generating transition paths that are likely to be feasible. The overall approach to the problem is defining a fitness function that can estimate how likely is that a transition path is feasible and how easy is it to generate an input sequence to trigger it. Then a GA is used to generate such transition paths. The results are compared to randomly generate transition paths with the same characteristics.

After addressing the path feasibility issue in EFSMs (Chapter 4), FSM based testing methods like UIOs can be used to test EFSMs. Having addressed the UIO generation problem for FSMs (Chapter 3), next we discuss (Chapter 5) how to combine the approaches in Chapters 3 and 4 in order to define UIO like state verification sequences for EFSMs and use GAs to generate them.

2.6 Communicating finite state machines (CFSMs)

Some systems are more naturally modelled as a set of FSMs (or EFSM) that interact by passing messages, and such FSMs (or EFSM) are known as communicating finite state machines (CFSM). A CFSM is like an FSM (or EFSM) but with input queues.

If we consider M as a set of CFSMs $M_1, ..., M_n$ then M_i denotes a CFSM $(S_i, s_{i1}, \lambda_i, \delta_i, X_i, Y_i)$ with an implicit input queue where $S_i = \{s_{i1}, s_{i2}, ..., s_{im_i}\}$. Transitions of a CFSM M_i are known as local transitions. When M receives an input x from the environment or another CFSM, x enters the input queue of M_i. It is assumed that M is deterministic. The output from one CFSM can trigger a transition in another CFSM and only the final output to the environment can be observed.

A CFSM is minimal if the corresponding FSM is minimal. Each M_i will be considered to be minimal, as there are standard algorithms for minimising FSMs. Only completely specified CFSMs are considered. M is assumed to be deadlock and livelock free as well and there are automated model checker tools that can verify this for formally specified systems [Baya 05].

The global state of M is defined by the states and the contents of the queues of M_i. A global state is said to be stable when all the queues of the M_i are empty. The state of an individual CFSM is referred to as a local state.

When some value x is fed to M while M is in a stable global state σ, a sequence of local transition will be executed. The final local transition will output some value y to the environment and leave M in a stable global state σ'. This global transition of M can be denoted as $(\sigma, \sigma', x/y)$.

Under certain conditions a set of CFSMs $M_1, ..., M_n$ can be converted into an equivalent single FSM called the product machine. However if n_i denotes the number of states in M_i, the product machine has $O(\prod_i n_i)$ states and hence suffers from combinatorial explosion. Chapter 6 briefly outlines how new alternatives to state and transition testing of CFSMs using genetic algorithms can be used without the need to convert them to an FSM.

For more background on CFSMs and testing CFSMs refer to [Bran 83, Lee 96].

2.7 Genetic algorithms

A Genetic algorithm (GA) [Gold 89, Srin 94] is a heuristic optimisation technique which derives its behaviour from a metaphor of the processes of evolution in nature. GAs have been widely used in search optimisation problems [Gold 89]. GAs and other meta-heuristic algorithms have also been used to automate software testing [Jone 98, Jone 96, Parg 99, Mich 01, Trac 00]. GAs are known to be particularly useful when searching large, multimodal and unknown search spaces. One of the benefits of GAs is their ability to escape local minima in the search for the global minimum.

Generally a GA consists of a group of individuals (population of genomes), each representing a potential solution to the problem in hand. An initial population with such individuals is usually selected at random. Then a parent selection process is used to pick a few of these individuals. New offspring individuals are produced using crossover, which keeps some of their parent's characteristics and mutation, which introduces some new genetic material. The quality of each individual is measured by a fitness function, defined for the particular search problem. Crossover exchanges information between two or more individuals. The mutation process randomly modifies offspring individuals. The population is iteratively recombined and mutated to evolve successive populations, known as generations. When the termination criterion specified is satisfied, the algorithm terminates. A flowchart for a simple GA is presented in Figure 2.3.

There are many different types of GAs, but they all share the basic principle of

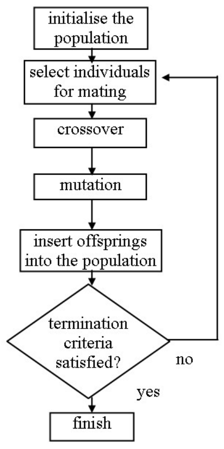

initialise the population
select individuals for mating
crossover
mutation
insert offsprings into the population
termination criteria satisfied?
no
yes
finish

Figure 2.3: Flowchart for a basic GA

having a pool (population) of potential solutions (genomes) where some are picked using a biased selection process and recombined by crossover and mutation operations. An objective function, known as the fitness function, defines how close each individual is to being a solution and hence guides the search.

When using a GA to solve a problem the first issue that needs to be addressed is how to represent potential solutions in the GA population. A genotype is how a potential solution is encoded in a GA, while the phenotype is the real representation of that individual. There are different representation techniques, the most common being binary and characters. Gray coding is a binary representation technique that uses slightly different encoding to standard binary. It has been shown [Whit 99] that Gray codes are generally superior to standard binary by helping to represent the solutions more evenly in the search space.

The first step in a GA involves the initialisation of a population of usually randomly generated individuals. The size of the population is specified at the start. Every individual is evaluated using the fitness function. When ranking is used the population is sorted according to the fitness value of the individuals. Then each individual is ranked irrespective of the size of its and its predecessors fitness. This is known as linear ranking. It has been shown that using linear ranking helps reduce the chance of a small number of very fit individuals dominating the search leading to a premature convergence [Beas 93a].

An important part of the algorithm is parent selection. A commonly used technique is the roulette-wheel selection. Here the chance of an individual being selected is directly proportional to its fitness or rank (if linear ranking is used). Hence the selection is biased towards fitter individuals.

A genome is made up of one or more chromosomes, each representing a parameter in the fitness function. In some of the literature the genome is referred to as a chromosome and genes refer to what we call chromosomes, but here we use chromosome as a part of a genome and gene as the building block of a chromosome.

The most common recombination technique used is crossover. During crossover the genes of the two parents are selectively used to create one or more new offsprings. The simplest crossover is known as single point crossover [Beas 93a]. For example Figure 2.4 shows how a single point crossover is applied to two parent chromosomes where two new child chromosomes are produced. There is also multiple point crossovers [Beas 93b]. In this thesis single point crossover is used with a randomly generated crossover point as used in [Mich 96].

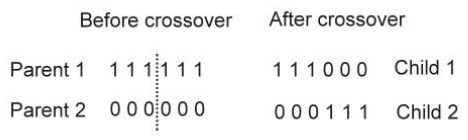


Figure 2.4: Example of crossover

Mutation is applied to each individual after crossover. It randomly alters one or more genes known as single point and multiple point mutation respectively [Gold 89]. Not all individuals are mutated. A pre defined mutation rate (typically the reciprocal of the chromosome length) is used to determine if mutation will be performed. A single point mutation with randomly selected point is used in this thesis as in [Mich 96].

There can be different termination criteria for a GA depending on the fitness function. If the fitness function is such that a solution would produce a specific fitness value, which is known, then the GA can terminate when an individual with such fitness is generated. However in many cases this is not known therefore the GA must be given other termination criteria. Such a criterion can be the specification of a maximum number of generations after which the GA will terminate irrespective of whether a solution has been generated. Another commonly used termination criterion is population saturation. After the fitness of all or some of the individuals in the GA population has not increased for a number of generations, it is assumed that a peak of the search space has been found that cannot be escaped. Usually a combination of these termination criteria are used. In this thesis we use all three but keep the criteria consistent throughout each set of experiments.

GAs are proven to perform well in unknown search domains. They have been widely used to help in NP-hard problems. There has been some interest in generating test case using GAs for a variety of systems and even some work on generating FSM based test cases [Guo 04]. In this thesis we use the test case adequacy estimation algorithms designed in attempt to guide GAs to generate the required test cases.

Chapter 3

Unique input/output sequence generation for FSMs

3.1 Introduction

The primary contributions of this chapter are showing how UIO generation can be formulated in terms of an automated search problem and describing an approach to automate UIO generation using genetic algorithms. This chapter demonstrates that UIO generation can be reduced to an automated search problem and presents results from an empirical study of this approach.

As outlined in Chapter 2, conformance testing is of particular interest when testing systems specified using FSMs. The generation of efficient and effective test sequences is very important in conformance testing. Test sequences can be generated using formal strategies like Unique Input Output Sequences (U-method), Distinguishing Sequences (D-method) and Characterizing Sets (W-method). The U-method is used in FSM testing for the following reasons [Sidh 89]. To implement the U-method a machine does not need to be completely specified. The UIOs can detect state transfer faults while coverage criteria like Transition Tours do not attempt this since they do not verify the final state of a transition sequence. UIOs lead to shorter test sequences than those produced using characterizing sets and the W-method also relies on a reliable reset for the FSM. There exist FSMs with UIOs for every state but no Distinguishing sequence. Practitioners report that in practice many FSMs have UIOs [Aho 91] for all states.

Ideally complete test suites are produced that would distinguish any faulty implementation given that it does not have more states than its specification. However, often this is not feasible because these methods rely on FSM with certain characteristics that cannot always be guaranteed. Work on generating complete test suits relies on either a distinguishing sequence (DS) being present in an FSM [Gone 70, Henn 64, Ural 97, Hier 02], the existence of a reliable reset in the FSM [Chow 78] or generation of test sequences of at least exponential (in terms of the number of states) length [Reza 95]. These issues will be later discussed in this chapter. Hence generating incomplete test suites has been of interest.

This chapter focuses on the U-method for test sequence generation [Sabn 88] where unique input/output (UIO) sequences for each state have to be generated. The problem of generating such sequences is known to be NP-hard [Lee 96]. While a random algorithm could be used it does not always produce acceptable results. Representing test sequence generation as a search problem with a specified fitness function gives the opportunity for algorithms known to be robust in searches of unknown domains, such as genetic algorithms [Gold 89], to be used. Generating test sequences using such algorithms could provide a computationally easy solution that produces good results as shown by [Guo 04].

One of the primary contributions of this chapter is the proposal for a more computationally efficient and yet effective method of generating UIO sequences. The proposed method also does not suffer from the usual restriction of some test sequence generation methods (D-method and W-method for example) where only fully specified

FSMs can be considered. The generated UIOs can be used for partially or completely specified FSMs that in turn can be used in generating a test sequence using the U-method. As a result weak conformance testing can be applied to partially specified FSMs without having any completeness assumption.

In order to minimise manual testing and hence software production costs and speed the process up, automation is necessary. Automation has been widely used in testing and test data generation [Sabn 88, Shen 89, Jone 98, Jone 96]. Automating the generation of UIO sequences can contribute to this.

3.2 UIO sequence generation

The problem of constructing UIO sequences is known to be NP-hard [Lee 96]. While a random search algorithm would be cheap to implement, it does not always produce acceptable results. Representing UIO sequence generation as a search problem with a specified fitness function gives the opportunity for algorithms known to be robust in searches of unknown domains, such as GAs, to be used.

A UIO for a given state s of an FSM is an input/output sequence that labels a sequence of transitions from s, but does not label a sequence of transitions from any other state. The UIOs considered in this work do not contain transitions unspecified in the FSM specification. This allows for weak conformance testing of partially specified machines. The proposed method uses GA search in an attempt to generate a UIO sequence for each state of a given FSM. A fitness function directs the search. The fitness function estimates how likely it is that a given transition sequence is a UIO sequence without actually verifying that it is one. For an input sequence of size l for a given state in an FSM with n states the fitness function used is of $O(l)$ complexity while a UIO verification algorithm would be of $O(nl)$.

Previous work [Guo 04] has shown that a GA may be used in the generation of

UIOs using a state splitting tree. A state splitting tree is a rooted tree that is used to construct adaptive distinguishing sequences or UIOs from an FSM. The fitness function encourages candidates to split the set of all states (in the root) into more discrete units (that share the same input and output characters). Hence the fitness function guides the search to explore potential UIOs by rewarding the early occurrence of discrete partitions while penalising the length of the sequence. The previous work differs from that described here in three important ways: (1) It used a more computationally intensive fitness function (based on generating the state splitting tree [Lee 96] and thus considering all states of the FSM); (2) It was evaluated only on relatively small FSMs; (3) Only completely specified machines were considered. In the work described in this chapter the fitness function is simpler and computationally easy to compute, and it also generated UIOs for partially specified machines.

3.2.1 Defining UIO generation as a search problem

When searching for a solution using genetic algorithms an efficient way must be defined to distinguish between potentially good and potentially bad solutions. A fitness function has been defined in order to represent the UIO sequence generation as a search problem. The fitness function determines how suitable a given transition sequence is to be a UIO sequence.

In order to verify if an input sequence would produce a UIO an algorithm with complexity of $O(nl)$ has to be executed where n is the number of states of the FSM and l is the length of the input sequence. Instead the proposed fitness function has complexity of $O(l)$, and this fitness function aims to reward sequences that are likely to be UIOs. Picking a less computationally complex algorithm for the fitness function is important since the algorithm can be executed several times for each state.

A transition ranking process is completed first before the fitness function is ready to be used. This process ranks each input/output pair of the specification machine

Start state	Input / Output	End state	Rank
1	a / 0	1	1
1	b / 1	2	1
2	a / 1	2	0
2	b / 1	3	1
3	a / 0	2	1
3	b / 0	1	0

Table 3.1: Transition table for the FSM from Figure 1 with I/O rankings

according to how many times it reoccurs in the transition table (a table with all the transitions of a machine) of the machine. A pair that occurs only once gets the lowest rank, a pair that occurs twice is ranked next etc. Pairs that have the same number of occurrences in the transition table get the same rank. For example Table 1 shows a ranked transition table for the FSM from Figure 2.1 (M_0).

It is important to note that execution costs for different transitions are assumed to be equal. Also equally ranked pairs are assumed to have similar ability to construct valid UIOs. Where this does not hold it would be straightforward to introduce extra information into the fitness function without increasing the complexity of the algorithm.

The fitness algorithm used in this chapter rewards a potential solution according to the ranks of the input/output pairs the sequence contains. The fitness function reflects the belief that the more low ranked transitions a sequence contains, the more likely it is to define a UIO. Some reported experiments in Section 3.3 investigate this claim. In fact if there is an input/output pair that is unique, then it automatically becomes a UIO, identifying the state from which the transition initiates. This fitness function however does not account for infeasible test sequences if partially specified FSMs are considered. Input characters testing unspecified transitions could result in unexpected behaviour of the IUT. Hence this fitness function works only for fully specified machine and in order for a partially specified machine to be used a completeness assumption has to be made. Below we explain how the fitness function can be adapted for partially specified machines.

Now consider the FSM M_0 in Figure 2.1. Using the fitness function defined above the fitness of an input/output sequence would be the sum of the ranks assigned to the input/output pairs it is composed of. If we consider the sequence $a/0, b/1$ as a potential UIO for $s1$ of M_0 and use the ranking provided on Table 1, a fitness value of 2 will be derived. This sequence is not a UIO as the same sequence could be executed and the same output observed from $s3$ of M_0. On the other hand while the sequence $b/1, a/1$ is considered from $s1$ a fitness value of 1 could be derived. This sequence is a UIO and its fitness value reflects the higher chance of it being a UIO compared to the sequence before that.

Not all FSMs are completely specified and protocols systems are typically partially specified [Lee 96]. In strong conformance testing assumptions on how the non core transitions are to be treated are made hence converting the machine into a completely specified FSM. For example one scenario is to add a transition with null output that stays in the same state. An alternative completeness assumption may be that if a transition is not in the core, then the machine makes a transition to an error state and outputs an error symbol. The missing transitions are treated as being 'do not cares' in weak conformance testing. The implementation is only required to have the same core behaviour, and can be arbitrary or undefined for the missing transitions.

Further refinements to the fitness function allow it to work for partially specified machines. This could facilitate weak conformance testing without a completeness assumption. For the purpose a simulator of the specification FSM was constructed. The FSM simulator (λ^* and δ^*) is a lightweight version of the IUT that only determines if a test sequence is feasible from a given start state and if not it indicates how close it came of being feasible. If an input character from a sequence represents an infeasible transition from a given state the input is ignored by leaving the FSM in the

```
$validValue := 0$
$strengthValue := 0$
$S_k := S_{UIO}$
If ($l = 0$) then return $\emptyset$
For($i := 1$ to $l$) //for all the inputs/characters
  $S_m := S_k$
  $y_i := \lambda(S_k, x_i)$
  $S_k := \delta(S_k, x_i)$
  If ($S_k \neq \emptyset$)
    //There is a transition with this input
    $validValue := validValue + 1$
    $strengthValue := strengthValue + r_{S_k,x_i}$
  EndIf
  Else
    $strengthValue := strengthValue + penaltyValue$
    $S_k := S_m$
  EndElse
EndFor
return $l - validValue + strengthValue$
```

Figure 3.1: UIO fitness algorithm

same state and then the next input character in the sequence is considered. Hence the whole input sequence under consideration can be evaluated by the fitness function even when an infeasible character has been reached. The fitness of infeasible input sequences is penalised according to how close the sequence came to be valid, while valid sequences are not penalised at all. The algorithm for the fitness function proposed is presented in Figure 3.1. The parameters involved are as follows: S_{UIO} is the test state; x is a single input character; y is single output character; l represents the length of the input sequence; r is the set of transition rankings where $r_{s,x}$ represents the rank for the transition initiating from state s with input x; and $penaltyValue$ is a penalty constant or function that penalises the fitness when an unspecified transition is triggered.

The test sequence generated like this would enable strong conformance testing with a less restrictive completeness assumption and weak conformance testing without any completeness assumption for a partially specified machine.

The whole process of searching for a UIO for each state of a given FSM can be easily automated as only the transition table of the FSM is required.

A GA using this fitness function is directed towards generating input sequences that contain mostly input/output pairs with lower frequency in the transition table corresponding to feasible transitions (in the specification). The fitness function represents the search for a UIO sequence as a function minimisation problem so an input sequence with a lower fitness value is considered to be more likely to form a UIO sequence since it is made up of more low ranked transitions.

3.2.2 Input sequence representation and GA

Generating a UIO sequence for a given state of an FSM would involve finding an appropriate input sequence that generates a unique output sequence. A specification simulator of the FSM can be used to simulate a transition path and generate the corresponding output. Hence a genotype representing a potential solution for a given state will only need to encode an input sequence.

A phenotype representing a sequence of characters can easily be represented as a genotype made of chromosomes for each character. Then each character can be represented as it is or encoded in binary notation. As described in Chapter 2 the classic GA approach would be to encode the characters in binary, but both methods could be applied to this problem. The GA tool used for UIO generation [Derd 02] supported only binary, hence that was the method of choice. Also in an attempt to reduce premature convergence in the population Gray coding was used instead of standard binary encoding [Whit 99].

A type checking process could be used to discard genotypes that do not represent valid phenotypes. When using binary representation the information that it translates to should ideally be in increments of the power of 2. Hence an input for an FSM with binary alphabet can be presented with a single digit in binary, an input with input alphabet of size 4, with 2 binary digits, etc. However there is a problem if the FSM considered has an input alphabet of size that is not a power of 2. In such cases a special type checking must be performed on all chromosomes within a genotype considered for fitness evaluation. The essence of this type checking is to ignore those binary combinations that do not translate to input characters from the input alphabet of the FSM considered. In the cases where a genotype is produced with invalid chromosome(s), the gene recombination, or generation in the case of the initial population generation, is repeated until a genotype where all the chromosomes are valid is produced.

This could potentially affect the speed of the algorithm as the input alphabet of the FSMs considered increases. However this was not evident in the experiments reported in Section 3.3. Sometimes the size of the input alphabet for an FSM is slightly bigger than its optimal binary representation (e.g. input alphabet of 17 will necessitate the use of a binary string of length 5 just because of one extra input character and introduces 15 redundant binary combinations). In such cases alternative binary to character translation techniques can be used [Mich 96] that distribute the number of valid characters and reduce the number of redundant binary combinations, but optimising this part of the generation algorithm is not the focus of this chapter.

The fitness function is designed so that it can compare only input sequences of the same length. For testing efficiency shorter sequences are more desirable, however we chose to separately consider the problem of having a fitness function and data representation that effectively addresses both the problem of UIO sequence generation and the length of such a sequence generated simultaneously.

Sequences of various lengths can be represented in binary for the GA in two ways. The first way is to simply have genotypes of different lengths encoding input sequence of different lengths. In this case the problem of how to apply the genetic recombination techniques has to be considered. Some work has been done on variable length genotype recombination, however these methods are very domain specific and no generic form is available [Gold 93]. A different approach is to encode different length input sequences by using the same length genotypes. This could be done by introducing a reset or sequence termination character to the sequence input alphabet. When such a character is reached in a sequence, the remaining characters encoded in the genotype will be ignored. In both situations the fitness function will favour shorter sequences to longer ones as they are likely to get a lower fitness value because of the fewer transitions involved. Initial experiments found that such a fitness function would always favour a single character sequence with just a reset character. Hence in order to generate a minimal UIO a set of generation attempts were made with gradually increasing sequence size.

3.2.3 Generating UIOs using genetic algorithms

After a fitness function and a phenotype representation technique are defined a GA can be used to find UIOs for all the states of an FSM. Verifying whether an input sequence is a valid UIO for a given state of an FSM is computation intense - $O(nl)$ for sequences of length l and n state FSM. So after a GA search stops, instead of checking if all the population individuals of the GA are UIOs only the sequence with the best fitness is considered. The result need not be a UIO sequence since not all FSMs have UIOs for all states or the GA might have converged prematurely i.e. the search might have converged to a local minimum. To increase the confidence that the input/output sequence found is the minimal length UIO for any given state i.e. a global minimum in the search space has been reached, the GA should be executed a number of times and only the best result kept.

For every GA execution the initial population is a set of randomly generated input sequences (genotypes). The corresponding output can be obtained from the FSM simulator generated from the FSM specifications (transition table). Each generated input sequence is type checked to see whether it represents a specified sequence of input of characters for the FSM under test. If not a new sequence is randomly generated until the initial population consists entirely of specified input sequences. The fitness is evaluated only for valid sequences. Hence any repeated attempts to generate sequences are not counted as fitness evaluations. The crossover and mutation operators recombine the selected genotypes in such a way that input sequences representing specified transitions with lower ranked input/output pairs are rewarded. Input sequences that represent some unspecified transitions, specified transitions with higher ranked input/output pairs or a combination of both would be rewarded less and penalised.

An example of how the GA recombination operators can help in this search follows. Lets consider an FSM M' for which the sequence $a/1, b/0, c/1$ is a UIO for

s_1. Assuming that *abc* is the only minimal UIO for s_1 lets take *aab* and *cbc* as two potential solutions in the population of the GA searching to find that UIO. Recombining these two sequences using a crossover at the first point would generate the necessary solution *abc*. Alternatively a crossover at the second point would generate the sequence *aac* that after a mutation at the second point can again produce the required *abc*.

The GA for every search terminates either after a set number of recombinations or if the population gets saturated with the same solution and does not improve for a number of generations. The lowest possible value for the fitness function cannot be negative but otherwise is unknown. Hence the GA cannot be set to terminate after an optimal solution is found. The only exception is when a single input character represents a UIO, then the fitness value evaluates to 0 and the GA terminates. Hence the GA currently used might have generated a solution much earlier than it actually terminates, but we have not yet attempted to optimise this aspect of the GA. Further work will aim to improve the fitness function and the generation algorithm so that fewer GA cycles are necessary before a solution is found.

3.3 Experiments

Most FSM examples available in the literature are not very large. A set of relatively small real FSM systems exists that is used for benchmarking purposes [LGSy 91]. This set can be used to examine the effects of the UIO generation algorithm on small but real FSMs and in order to examine how it performs on larger FSMs a set of larger randomly generated FSMs was used.

The first set of experiments considers a set of 11 real FSMs (Table 3.2). The FSMs ranged in size from 4 to 27 states and 10 to 108 transitions. The second set of experiment was conducted on a set of 23 randomly generated FSMs (Table 3.3).

FSM	States	Transitions	Inputs	Outputs
dk15	4	32	8	11
mc	4	32	8	8
bbtas	6	24	4	4
beecount	7	51	8	4
dk14	7	56	8	15
dk27	7	14	2	3
shiftreg	8	16	2	2
dk17	8	32	4	5
lion9	9	25	4	2
dk512	15	30	2	4
dk16	27	108	4	5

Table 3.2: List of the 11 real FSM examples used

These FSMs ranged from 5 to 360 states and 14 to 901 transitions in size [1]. Both sets consisted only of deterministic, strongly connected, and minimal but not necessarily completely specified finite state machines.

A breadth first search (BFS) algorithm can be used to enumerate through all possible input sequence combinations. By verifying each combination we can exhaustively (up to a fixed input sequence limit) find all the minimal length UIOs (within that limit). This approach would require each input sequence to be verified using a UIO verification algorithm ($O(nl)$). On the other hand the GA approach presented in this chapter verifies only one input sequence at the end of a GA execution. For that reason it is difficult to present a precise comparison of effort between the GA and a BFS algorithm but a rough figure, biased towards the BFS is presented.

The minimal UIOs found for all the states of an FSM by the two GAs and random algorithm were considered. The shortest UIO for each state was listed. The longest UIO in this list was used as an indicator of what maximum length input sequence a BFS algorithm would be expected to generate for a given FSM in a worst case scenario.

[1]The experiments were carried out on FSMs with at most 360 states due to the prototype tool being limited to FSMs with no more than 1000 transitions. This restriction was due to a combination of Java features and the tool design.

FSM	States	Transitions	Inputs	Outputs
1	5	14	4	2
2	10	33	4	2
3	20	51	4	2
4	39	87	4	2
5	50	136	4	2
6	73	177	4	2
7	90	218	4	2
8	98	250	4	2
9	113	296	4	2
10	132	316	4	2
11	158	393	4	2
12	180	450	4	2
13	203	498	4	2
14	209	553	4	2
15	227	568	4	2
16	244	611	4	2
17	264	658	4	2
18	291	765	4	2
19	305	771	4	2
20	311	765	4	2
21	323	809	4	2
22	347	856	4	2
23	360	901	4	2

Table 3.3: List of the 23 randomly generated FSM examples used

This figure is compared to the number of fitness evaluations (including unsuccessful UIO generation attempts) by the GAs and random UIO generation (the 2 GAs and random were given the same effort in terms of fitness evaluations). Figure 3.2 shows the *difference* between these two figures for all 23 randomly generated FSMs. As some of the BFS input sequence variations go into billions Figure 3.3 shows the same information but filtering the 4 worst estimated BFS effort FSMs. From the graph it is clear that for many FSMs the BFS algorithm could have been more efficient to use. This is because of the mainly short UIOs found for many of the FSMs. However the graphs also indicate that the BFS is much worse in some cases, where the GA performed well.

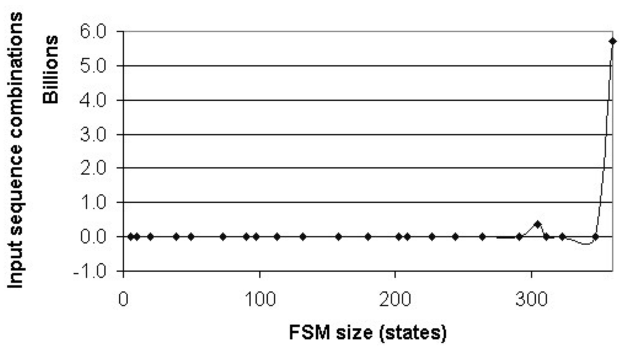


Figure 3.2: Difference in effort between worst case BFS and current GA results in attempt to find all UIOs of an FSM

It was expected that when small FSMs are considered the real advantage of the new method cannot always be observed over the random test sequence generation method. However as the size of the FSMs considered increases the proposed method is expected to outperform the random method. Since BFS is not feasible for those FSMs where the benefits of using GA are likely to be observed, BFS was not included in the experiments.

Some results justifying the UIO generation algorithm choice are presented first.

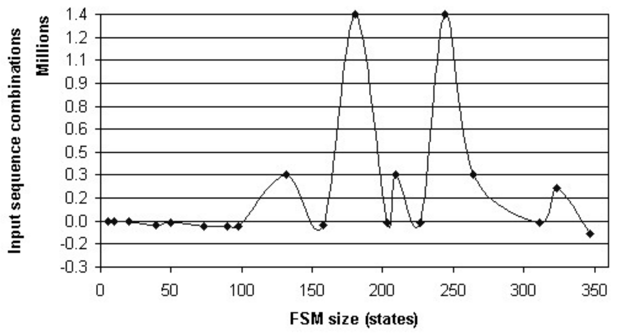


Figure 3.3: Difference in effort between worst case BFS and current GA results in attempt to find all UIOs of an FSM - 4 worst performing FSMs for BFS removed from graph

Then the actual performance of the algorithm is compared with the random generation algorithm. The reason for using two different GA types was to determine whether the slightly different heuristics generate different results. The first GA used a single point crossover and mutation while the second used a complex multiple point crossover and mutation. In general the second GA tended to find a solution slightly faster than the first GA, but they produced the same results. Hence for most FSMs the two GAs show identical performance.

3.3.1 UIO generation process

For any successful heuristic search it is imperative that a fitness function is selected that guides the search correctly towards a solution. In the search for UIOs the degree of difference (DoD) metric can be used [Naik 95]. A DoD compares the output sequence β generated by an input sequence α from state s_i with the corresponding output sequence from state s_j. We can extend this notion and instead of comparing the output sequence of s_i only to that of s_j, where s_j is just another state, we can compare it to all states apart from s_i. We sum all the individual DoD values into one cumulative DoD for a given UIO. This process is of the same complexity as the UIO verification algorithm - $O(ln)$. In this chapter we refer to this cumulative DoD value.

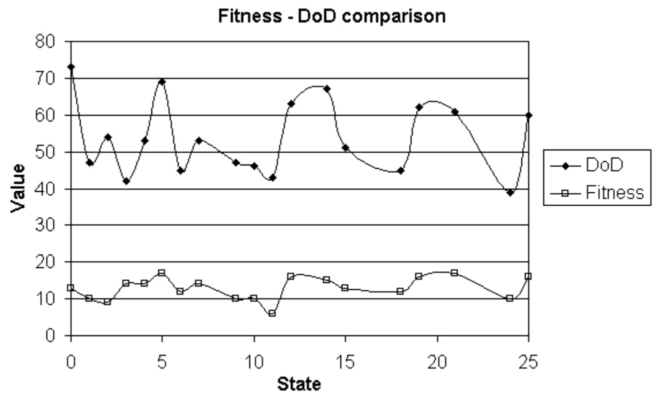


Figure 3.4: Fitness function value and DoD for a set of UIOs for FSM dk16

Figure 3.4 has two graphs representing the DoD and fitness values of 19 input sequences for the $dk16$ FSM, the largest from the set of real FSM examples. These 19 input sequences represented 19 UIOs for different states of that FSM. The vertical scale of the graph represents the fitness and DoD values while the horizontal represents the state of the UIO. It can be seen how the shape of the fitness function closely follows the shape of the DoD, except for the extent of the actual rises and falls of the DoD. This indicates that the fitness function, although not calculating the DoD for a given input sequence, can serve as a rough estimate of which input sequences are likely to have higher DoD and hence are likely to be UIOs (and more robust UIOs). Therefore the fitness function is likely to be directing the search in a positive direction without the full expense of calculating the DoD.

As mentioned before the UIO generation process used involved verifying whether a given input sequence is a UIO for a given state. After a GA has terminated only the highest ranked element in the final population is verified to see whether it represents a UIO because of the computational complexity involved with this checking process. Verifying an input sequence as a UIO is the most expensive part of the algorithm but it would not make sense to verify only the top ranked individual of a population if

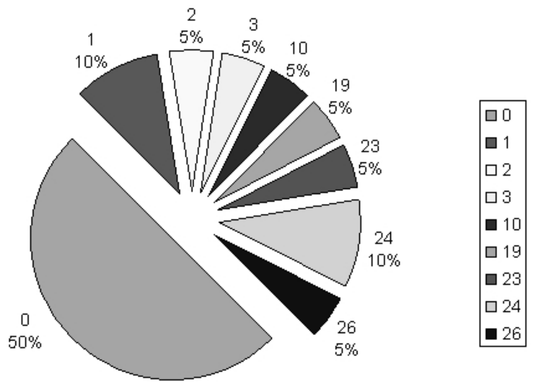


Figure 3.5: Positions in the GA population where the first valid UIO was found for each state of FSM dk16 (the largest of the real FSMs)

such individuals do not tend to be UIOs. Figure 3.5 represents the rank of the first element within the 20 terminated GA populations which generated UIOs for the $dk16$ FSM (the largest of the real FSMs). Half of the UIOs were found at the top, 0-th position of their corresponding GA population. The next highest ratio represented only 10% of the results. The rest of the real FSMs had even higher ratio of UIOs found at the top, 0-th position of their corresponding GA populations. This suggests that we lose little by verifying only the top ranked individual but we reduce the complexity of the whole UIO generation process since we repeat the search if a UIO is not found. It is simple to adapt the algorithm so that it checks all elements of the final population or some fixed proportion of this.

3.3.2 UIO Generation

A set of experiments involving UIO generation were run using the two sets of FSMs. Two slightly different GAs and a random search algorithm were used for every FSM. After each UIO generation attempt a simple algorithm was used to determine whether the sequence is indeed a valid UIO and does not contain unspecified transitions. The GAs used a single, ranked population where fitter genotypes are added by removing the genotypes with the lowest rank. The genotype selection was done using roulette wheel selection [Srin 94]. Gray coding [Mich 96] was used as the chromosome representation technique. The recombination operators used were uniform crossover and uniform binary mutation with mutation rate of 0.05. The first GA used the classic genotype recombination while the second GA used a chromosome recombination where each input character for a transition sequence is represented as a separate chromosome. The second GA performs recombination independently on each character of the input sequence. The termination criteria were population saturation or up to 10,000 fitness evaluations. A UIO generation attempt for a given state in the FSM involved no more than 3 GA executions, for each of the sequence sizes (number of chromosomes) considered with up to 25 inputs. The fittest phenotype after each GA termination was considered as a potential UIO sequence. As soon as a valid UIO was found for a given state in the FSM the search moved to the next state. For the randomly generated FSMs no more than 15 GA executions were considered for each sequence size up to 45 inputs because these FSMs are larger and we expected that more effort would be required to generate UIOs.

After sequences were generated with the two GAs, random sequence generation was applied. After a number of random input sequence generations, within the FSM input alphabet constraints, the sequences were ranked and the fittest one was checked to determine whether it was a UIO. The number of random generation attempts (to generate a UIO) for a state of the FSM used was equal to the average number of attempts it took the GAs to generate a UIO for that particular state. Every attempt to generate a sequence for a given state was repeated for sequence sizes ranging from the shortest to the longest UIO sequence found for this state by the GAs. The random search was given at least the same computational power in terms of number of fitness evaluations and UIO verification attempts.

Figure 3.6 and Table 3.4 show the results of the UIO generation algorithm conducted on the set of 11 real FSMs. For each FSM two different types of GA algorithms and a random generation algorithm were executed in an attempt to generate UIOs for each state.

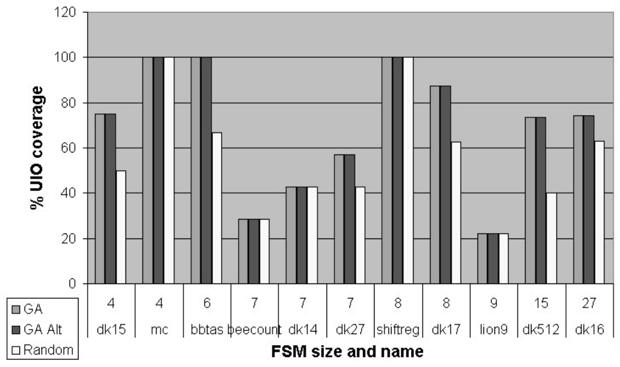


Figure 3.6: Percentage state coverage in UIOs generated by GA compared to random algorithm. Results for real FSMs

Some of the FSMs considered have a very small number of states. For such FSMs a single input character might represent a UIO. In such cases it is obvious that the random algorithm will be effective. For example all the UIOs in the *mc* FSM were of length 1. It is also important to note that not all FSMs have UIOs for all states. For example the *lion*9 and *becount* FSMs have UIOs for only 2 of their states, and they were found by the UIO generation algorithms. The number of UIOs generated were compared to results reported in [Sun 98, Sun 01]. FSMs $dk14-17$ and $dk512$ were reported to have the same UIO state coverage as we found. In [Sun 98] the $dk16$ FSM was reported to have UIOs for 21 of its 27 states, however [Sun 01] reported that it only has UIOs for 20 states and we manually verified that. The GA produced UIOs for these 20 states. FSMs *mc*, *bbtas* and *shiftreg* had UIOs generated for all their states. This shows that for each FSM the GA UIO generation managed to find at least one UIO for all the states that had one. Also for most of the FSMs the GA based UIO generation outperformed the random generation generating up to 33%

FSM	States	GA %	GA Alt %	Ran. %	Diff.%
dk15	4	75	75	50	**25**
mc	4	100	100	100	0
bbtas	6	100	100	67	**33**
beecount	7	28	28	28	0
dk14	7	43	43	43	0
dk27	7	57	57	43	**14**
shiftreg	8	100	100	100	0
dk17	8	88	88	63	**25**
lion9	9	22	22	22	0
dk512	15	73	73	40	**33**
dk16	27	74	74	63	**11**

Table 3.4: Percentage state coverage in UIOs generated by GA compared to random algorithm. Results for real FSMs

better results. As expected not all the UIOs generated were minimal.

Now consider the experiments with (larger) randomly generated FSMs. Both GA search based UIO generation techniques performed better for all 23 randomly generated FSMs, sometimes generating UIOs for up to 62% more states than the random search. The two GAs produced identical UIO state coverage results. Figure 3.7 shows the number of states for which a UIO has been generated as a percentage of the total number of states of the FSM using the three methods. Figure 3.8 shows the same data but plots the difference in the percentage between the random search and the two GA methods. Here it appears that the difference between the GA and random algorithm increases as the size of the FSMs increases. Both graphs clearly illustrate the potential advantage of using GA search against random search for UIOs, when using the fitness function considered. It is important to remember that different FSMs have different properties. For example not all FSMs have UIO sequences for all their states. Hence the graphs are not very smooth. Again, not all the UIOs generated were minimal.

Another interesting result was that the average UIO sequence size was much

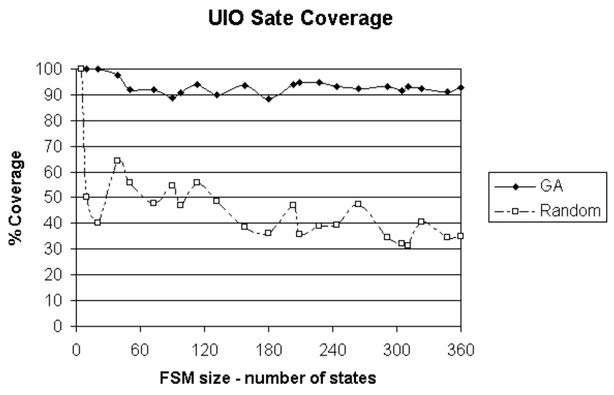


Figure 3.7: Percentage state coverage in UIOs generated by GA compared to random algorithm. The two GAs produced identical results. Results for Randomly generated FSMs

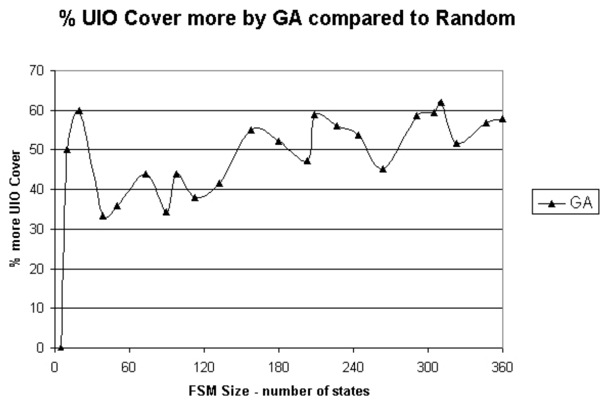


Figure 3.8: Percentage difference in UIOs generated by GA compared to random algorithm. Results for Randomly generated FSMs

shorter than expected as in the worst case the length of a UIO is exponential in terms of the number of states of the FSM [Lee 94]. In fact, most of the UIO sequences seem to be very short, even for larger FSMs. In comparison, a separating sequence is expected to be of size $n-1$ at most, but it has been observed that its expected size is of $O(log(n))$ [Trak 73]. Figure 3.9 shows the average UIO sequence length for each of the 23 FSMs using the GA methods and the random search. The graph does not seem to increase exponentially, but it actually seems to increase at a rate less than linear. Since most of the larger FSMs on the graph have state coverage as high as 95%, indicating that there is a small number of UIOs left to be found to achieve full state coverage, it seems that most of the UIOs tend to be very short.

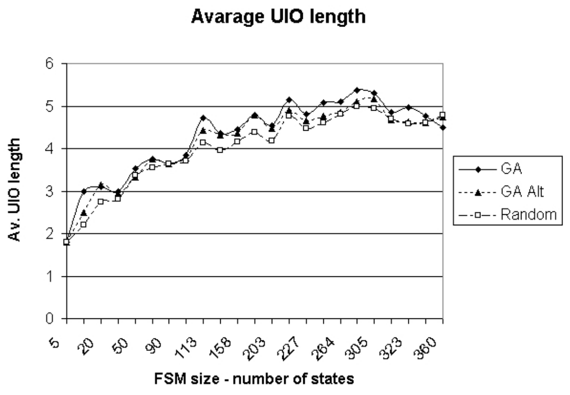


Figure 3.9: Average UIO size found for the Randomly generated FSMs

3.4 Summary

State verification is an important part of conformance testing for FSMs. UIO sequences are commonly used for state verification because of their advantages over the other methods. The problem of generating such sequences however is known to be NP-hard [Lee 96]. While a random algorithm could be used it does not always produce acceptable results. GAs have previously been used to generate UIOs for relative small and completely specified FSMs [Guo 04].

In this chapter we define the problem of finding UIO sequences as a search problem. We define a computationally efficient fitness function of $O(l)$ complexity for an input sequence of size l that is used to guide a GA. UIOs for both completely and partially specified FSMs were generated. This approach considers partially specified FSMs and generates UIOs that can also be used for weak conformance testing without completing the FSM.

We investigate the performance of a GA search for UIOs for an FSM using this fitness algorithm on a number of real and some larger randomly generated FSMs and report the results.

UIOs were computed using GA and random search. The experiment includes two groups of FSMs: a set of 11 real FSM specifications of small size; and a set of 21 randomly generated FSMs with up to 360 states. The fitness function appears to direct the search towards generating UIOs. The experiments show that the GA outperforms (up to 62% better) or is at least as good as a random search for UIO sequences. As the size of the FSMs increased the difference between the performance of the GA and random UIO generation also increased.

The results also show that the average UIO size tends to be small even for larger FSMs. Most of the UIOs found were no longer than 10 input/output pairs. Searching for UIOs using a breadth first search algorithm for some of the larger FSMs considered could run into billions of input sequence generations in a worst case scenario (judging from the minimal UIOs we have found for those FSMs). However breadth first search could be more efficient than GA for very short UIOs. This could suggest that breadth first search or even random search can be very useful for generating most of the UIOs, which are very short. GA search can subsequently be used to search for longer UIOs which are otherwise computationally difficult to identify using breadth-first search. This work can also be extended to help in test sequence generation for EFSMs

(Chapter 5) and CFSMs (Chapter 6) where the problem becomes much harder due to the complexities associated to internal variables and communication.

Chapter 4

Feasible Transition Path generation for EFSMs

4.1 Introduction

This chapter addresses the issue of finding feasible transition paths (FTPs) between two states for systems based on the EFSM model. A novel way of abstracting parts of the data in the EFSM in order to facilitate the generation of feasible transition paths using GAs is presented and evaluated.

Definition 4.1.1. A transition path (TP) represents a sequence of transitions in M where every transition starts from the state where the previous transition finished.

The chapter begins by outlining the problem. The approach used to search for FTPs is described next. A number of preprocesses that need to be executed once for every EFSM before it is analysed are then discussed. Then in Sections 4.4.3 and 4.4.5 the fitness functions for two different FTP search problem representation are described as well as the algorithms used to verify potential solutions. A description of the experimental strategy to collect the empirical evidence follows. An overview of the empirical evidence that examines the effectiveness of the feasibility estimating fitness function is followed by the FTP generation results. Finally, conclusions are drawn.

4.2 Problem

A system specified by an FSM or an EFSM is tested for conformance by applying a sequence of inputs and verifying that the corresponding sequence of outputs observed is that expected. Typically the machines that arise are complex and brute force testing is infeasible [Lee 96].

In EFSMs test sequence generation is more complex than it is for FSMs. In FSMs all of the paths of the directed graph are valid since there are no conditions on the edges and actions do not affect the graph traversal [Dual 00]. With EFSMs, however, the transition path depends on the result of the input parameter guard and the domain guard (as defined in Chapter 2). The result of the guard depends on the values of the internal variables and input declarations, which in turn can assume different values after each transition. Some transition paths might have no conditions, some might have conditions that are rarely satisfied and some transition paths will be infeasible. The existence of infeasible transition paths creates difficulties for automating the test generation process for EFSMs.

One way of approaching the test sequence generation problem is to abstract away the data part of the EFSM and consider it as an FSM on its own. However a transition sequence for the underlying FSM of an EFSM is not guaranteed to be feasible for the actual EFSM. Another way is to expand an EFSM to an FSM and then use the techniques used for FSMs. However this can lead to a combinatorial explosion.

Some recent work on generating feasible conformance test sequences for EFSMs was presented in [Dual 04]. It describes how to remove variable interdependencies between the actions and conditions of a certain class of EFSMs. The inconsistency removal problem, exponential in complexity, used there takes advantage of localised inconsistencies to reduce the algorithm complexity and bound it by the size of the subgraphs involved. After the inconsistencies removal, FSM based methods are used for test sequence generation. Our EFSM model is more general as we do not consider only linear constraints. [Dual 04] gives a comprehensive summary of all the important work in this field up to now. Test generation for EFSMs is still an open research problem [Lee 94, Dual 04]

The general problem of finding a (an arbitrary) feasible transition sequence for an EFSM is uncomputable, as is generating the necessary input sequence to trigger such a transition sequence. While a random algorithm could be used it does not always produce acceptable results. GAs have been used in related problems like generating UIOs for FSMs [Guo 04, Derd 06] and generating test input sequences for communicating FSMs [Derd 04]. Heuristic search techniques can be also applied to the FTP generation problem if a robust fitness function can be defined.

The form of EFSM used in this chapter is the Normal Form EFSM (NF-EFSMs) defined in Chapter 2. It is a general EFSM model that can often be obtained directly from SDL specifications [Hier 03]. For the remaining of this chapter we will refer to NF-EFSMs simply as EFSMs.

The objective of this work is to facilitate the generation of feasible transition paths in EFSMs and the necessary input to trigger them. This can be helpful in test data generation for EFSMs. In this chapter we focus on generating transition paths that are likely to be feasible. The overall approach to the problem is based around defining a fitness function that can estimate how likely it is that a transition path is feasible and how easy is it to generate an input sequence to trigger it. Empirical data is used to evaluate the effectiveness of this fitness function. Then a GA is used to generate such transition paths. The results are compared to randomly generate transition paths with the same characteristics.

4.3 Generating feasible transition paths (FTPs) for EFSMs

In this section we define a feasible transition path (FTP) and describe a fitness function that is intended to estimate how likely a transition path is to be feasible (Section 4.3.1).

Consider the problem of finding an input sequence that triggers a feasible transition path (FTP) from state s_i to state s_j of an EFSM M.

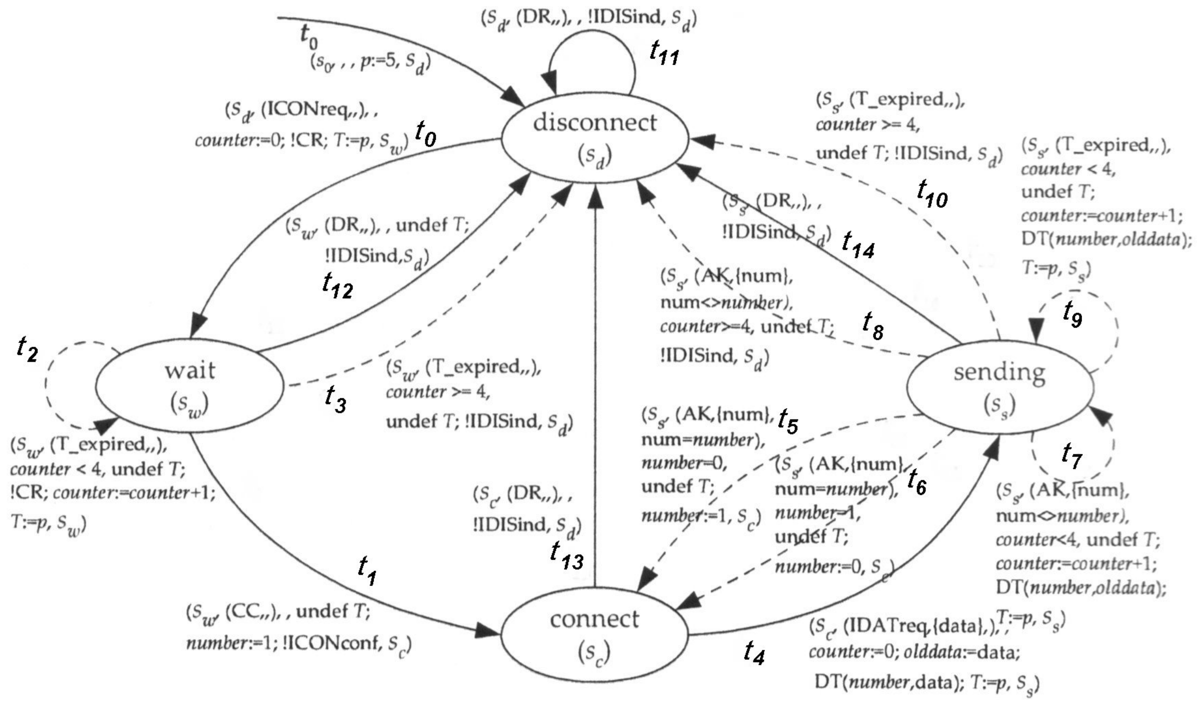
Definition 4.3.1. A forward feasible transition path (F-FTP) for state s_i of an EFSM M is a sequence of transitions initiating from s_i that is feasible for at least one combination of values of the finite set of internal variables V of M.

State identification and state verification sequences for an EFSM must trigger an F-FTP. Hence generating an F-FTP and an input sequence that triggers it for state s_i of EFSM M can help in finding a state identification or state verification sequence for s_i.

Definition 4.3.2. A backward feasible transition path (B-FTP) for state s_j of an EFSM M is a sequence of transitions ending at s_j that is feasible for at least one combination of values of the finite set of internal variables V of M.

A BF-FTP is a feasible transition path with specified both start state s_i and end state s_j. When we use FTP we refer to all three types.

In a transition path for FSMs each transition can be identified and thus represented by its start state and input (s_s, i). However with EFSMs this information is not sufficient because there can be more than one transitions sharing the same start state and input. Instead a transition t in an EFSM M can be identified from its start state, input declaration, input parameter, the input parameter guard and the domain guard



t0
(s0, , p:=5, Sd)
(Sd, (ICONreq,,), , counter:=0; !CR; T:=p, Sw) t0
(Sd, (DR,,), , !IDISind, Sd)
t11
disconnect (sd)
(Ss, (T_expired,,), counter >= 4, undef T; !IDISind, Sd)
t10
(Ss, (T_expired,,), counter < 4, undef T; counter:=counter+1; DT(number,olddata); T:=p, Ss)
t9
(Sw, (DR,,), , undef T; !IDISind, Sd)
t12
(Ss, (DR,,), , !IDISind, Sd) t14
(Ss, (AK,{num}, num<>number), counter>=4, undef T; !IDISind, Sd)
t8
t2
wait (sw)
(Sw, (T_expired,,), counter >= 4, undef T; !IDISind, Sd)
t3
sending (ss)
(Sw, (T_expired,,), counter < 4, undef T; !CR; counter:=counter+1; T:=p, Sw)
(Ss, (AK,{num}, num=number), number=0, undef T; number:=1, Sc)
t5
(Ss, (AK,{num}, num=number), number=1, undef T; number:=0, Sc)
t6
t7
(Ss, (AK,{num}, num<>number), counter<4, undef T; counter:=counter+1; DT(number,olddata); T:=p, Ss)
(Sc, (DR,,), , !IDISind, Sd)
t13
t1
(Sw, (CC,,), , undef T; number:=1; !ICONconf, Sc)
connect (sc)
(Sc, (IDATreq,{data},), , counter:=0; olddata:=data; DT(number,data); T:=p, Ss)
t4

Figure 4.1: Inres protocol as an EFSM M_1

$(s_s, i, P^i, g_{P^i}, g_D)$ (as defined in Chapter 2). g_{P^i} and g_D for a transition t in M can both be logical expressions and their results may depend on input parameter P^i of t and the values of some of the internal variables of M. Transitions sharing the same start state and input declaration can be classified according to their input guard predicate and domain guard predicate. To identify these for every set of transitions sharing the same start state s and input declaration i, the number of possible combinations of input parameter guard logical expression (input predicate branches) and the domain guard logical expression (domain predicate branches) is counted and a predicate dependency graph for state s and input declaration i can be constructed.

Consider the Sending state (S_s) in the EFSM M_1 on Figure 4.1. There are four transitions initiating from this state that share the same input declaration AK and input parameter num. A partial transition table for this state (Figure 4.2) represents all the outgoing transitions from state S_s of M_1 with the same input that differ only in their input parameter guards and domain guards. From the information in the table we can construct a predicate dependency graph for state S_s of M_1 with input declaration AK. The four leaves of the graph in Figure 4.3 represent the four different transitions sharing the same start state and input.

i	P^i	g_{P^i}	g_D
AK	num	$num = number$	$number = 0$
AK	num	$num = number$	$number = 1$
AK	num	$num \neq number$	$countr \geq 4$
AK	num	$num \neq number$	$countr < 4$

Figure 4.2: Partial transition table for transitions starting from S_s in EFSM M_1 (Figure 4.1)

Definition 4.3.3. A predicate dependency graph classifies all transitions from state s that share the same input, according to the input predicate guard (g_{P^i}) and domain guard (g_D) for each of these transitions.

Hence a transition in a transition path of an EFSM can be identified by its start state, input, g_{P^i} and g_D (s_s, i, g_{P^i}, g_D). The input parameter P^i is not required in order to be able to uniquely identify a transition in M. Note how in this case some transitions with different domain guards share a common input predicate guard.

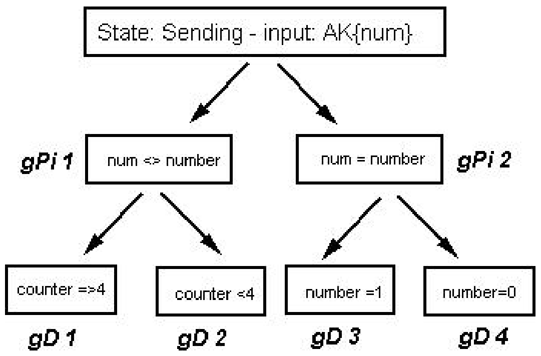


Figure 4.3: Predicate dependency graph for the transitions on Figure 4.2

Not all transitions in EFSMs have input parameter guards and domain guards and so transitions in an EFSM M can be categorised in the following way:

- **simple transitions** are those transitions that have *no* input parameter guard and *no* domain guard, $g_{P^i} = NIL$ and $g_D = NIL$.
- g_{P^i} **transitions** are those transitions that *have* input parameter guard but *not* a domain guard, $g_{P^i} \neq NIL$ and $g_D = NIL$.
- g_D **transitions** are those transitions that *have* a domain guard but *not* an input parameter guard, $g_D \neq NIL$ and $g_{P^i} = NIL$.
- g_{P^i}-g_D **transitions** are those transitions that *have* both an input parameter guard and a domain guard, $g_{P^i} \neq NIL$ and $g_D \neq NIL$.

The sequential operations *op* for a given transition t can consist of simple assignments and output statement. However in EFSMs besides output declarations there are also output functions that take a number of parameters and generate an output based on these parameters. An assignment statement in the *op* part of a transition would not generate any observable output. However such assignment statements in the *op* part of a transition can still contribute to the value of the output generated in a later transition if the assignment changes the value of one of the output function parameters. It can also affect the feasibility of the remaining transitions in the transition path and should also be considered.

Definition 4.3.4. An input sequence (IS) is a sequence of input declarations $i \in I$ with associated input parameters $P^i \subseteq P$ of an EFSM M.

Instead of using g_{P^i} and g_D notations together in order to identify a transition we can simply index all the leaves of the predicate dependency graph and use a single value.

Definition 4.3.5. A predicate branch (PB) is a label that represents a pair of g_{P^i} and g_D for a given state s and input declaration i. A PB identifies a transition within a set of transitions with the same start state and input declaration.

PBs can label conditional transitions and be used to help simulate the behaviour of a potential input sequence for an EFSM without the feasibility restrictions.

Definition 4.3.6. An abstract input sequence (AIS) for M represents an input declaration sequence with associated PBs that triggers a TP in the abstracted version of M.

In order for a sequence of input parameters to represent an abstract transition path that is feasible the conditions on each PB must be satisfied. This can lead to conditions on the values of some of the internal variables of the EFSM.

4.3.1 Fitness function

In order to achieve our objectives we require an easy to compute fitness function to guide the search for FTPs. Computing the actual feasibility of a transition path is too computationally expensive, so we need a method to estimate this.

Some transition paths consist of transitions with difficult to satisfy guards. The presence of *simple* transitions in a transition path makes this TP likely to be an FTP since there are no guards to be satisfied. The presence of g_{P^i} transitions could render a transition path infeasible because of its input predicate guard. However the conditions of this guard are more likely to be satisfiable than domain guards because the values of the input parameters $P^i \subseteq P$ can be chosen. When these conditions depend also on some internal variables $V' \subseteq V$ then such g_{P^i} transitions might not be easier to trigger than g_D transitions. In some cases the execution of a g_D transitions could require reaching its start state through a specific transition path. The feasibility of g_{P^i}-g_D transitions depends on both issues outlined above for g_{P^i} transitions and g_D transitions.

Hence the presence of g_D transitions and g_{P^i}-g_D transitions seem to increase the chance of a transition path being infeasible. Avoiding such transitions and encouraging *simple* transitions could increase the chance of generating FTPs. A fitness function that favours transition paths that start from s_i with less constrained transitions can be used to direct a search for F-FTPs from s_i. A fitness function that rewards transition paths according to how many states away from s_j they end and favours less constrained transitions can be used to direct a search for B-FTPs for s_j.

One important issue to consider is how to weight the fitness of transitions of different types. A table with the associated penalties for every transition in a transition path is presented in Figure 4.4. The penalty values shown are by no means definitive, but they aim to direct the search according to the transition type preference argument presented above. It has been assumed that $\neq$ is the easiest type of comparison operator to be satisfied while $=$ is the most difficult. Naturally, for some EFSMs exactly the opposite might be true in certain cases. More detailed analysis of the EFSM might lead to better guidance.

Operator	*simple*	g_{P^i}	g_D	g_{P^i}-g_D
$\neq$	0	1	2	4
$\geq$ or $\leq$	0	3	4	8
$>$ or $<$	0	4	5	10
$=$	0	6	7	14

Figure 4.4: Penalties for each condition in a transition

When there is more than one condition in a guard the penalty value associated with the guard depends on the logical operator between these conditions. With AND operator the sum of the penalties is used. For OR operator only the lowest penalty is used.

A transition ranking process is completed first. This process ranks each transition of the EFSM according to how many penalty points are assigned to the transition guards. A *simple* transition gets the highest rank, an g_{Pi} transition with one condition is ranked next etc. Transitions that have the same number of penalty points get the same rank.

The fitness algorithm used in this work rewards a potential solution according to the ranks of the transitions in the sequence. The fitness function reflects the belief that the fewer constraints a sequence contains, the more likely it is to define an AIS for FTP. When there is an AIS that triggers a transition sequence of only *simple* transitions, then it automatically becomes an FTP.

4.3.2 TP Representation

The two different FTP search problem representations in this chapter are based on different TP representation notation.

The first approach, called *Transition notation*, represents a TP using transition labels from the transition table. This representation is simple but does not guarantee that a list of transition labels represent sequential transitions. This notation represents an exact TP. The IS that might potentially trigger such a TP can be generated from the transition table.

The second approach, called *PB notation*, uses input declarations and PBs to define an AIS. The AIS represents one TPs, but the actual TP followed is determined at run-time.

The quality of all TPs generated using both representations is evaluated using an FTP verification process outlined in Section 4.4.4.

Each notation defines a search space with different shape and size. The size of the search space for transition notation and PB notation depends on the number of transitions and on the combined number of input declarations and PBs respectively. For large EFSMs with many transitions, the search space with PB representation is likely to be smaller. However for small EFSMs where the combined number of input declarations and PBs exceeds the number of transitions, the transition notation might present a smaller search space.

The work in this chapter does not focus on the shape of the search spaces of the example EFSMs we discuss, however the results of the experiments in Section 4.5.3 provide an indication of how GAs perform when searching such a space.

The fitness functions for the two different FTP search problem representations are discussed in detail in the next section.

4.4 Algorithms

4.4.1 Preprocesses

A number of process must be completed before the fitness function and FTP verification algorithm can be used for an EFSM. The four processes are shown on Figure 4.5, where $|T|$ is the number of transitions and n is the number of states of the EFSM (as defined in Chapter 2). The table lists the complexity of each process and which of the two FTP representations that process facilitates. The input enumeration and PB count processes are specific for the PB notation, while performing transition ranking and counting transition distances between all state pairs (ϕ matrix) is used for both PB notation and transition notation.

	Process	complexity	Used for PB	Used for transition not.
1	Transition ranking	$O(\|T\|.log\|T\|)$	✓	✓
2	Input enumeration	$O(\|T\|)$	✓	x
3	PB count	$O(\|T\|)$	✓	x
4	Calculating ϕ matrix	$O(n.\|T\|)$	✓	✓

Figure 4.5: Preprocesses used by the fitness and verification functions for FTP generation using the PB notation and transition notation. $|T| \geq n$ since we are looking at initially connected EFSMs

A *transition ranking* process is completed first before the fitness function can be used. This process ranks each transition of the EFSM according to how many penalty points are assigned to the transition guards. A *simple* transition gets the highest rank (i.e. lowest amount of penalty points), an g_{P^i} transition with one condition ($\neq$) is ranked next etc. Transitions that have the same number of penalty points get the same rank. This algorithm in essence sorts $|T|$ elements and has complexity $O(|T|.log|T|)$ where $|T|$ is the number of transitions in M.

A simple *input declarations enumeration* process is also necessary before a GA search can be started. This process enumerates all the unique input declarations $i \in I$ used in M and assigns numbers that in turn are used to represent each input declaration i in the GA phenotype. Here input predicates P_i are not considered as part of any input declaration i. This algorithm has complexity of $O(|T|)$ and can be executed in conjunction with the transition ranking process.

A *PB counting algorithm* is also used to assist in the GA phenotype representation. All the transitions that share the same start state and input declaration are enumerated and each assigned a PB index. For example, in Figure 4.6 four transitions share the same start state and input declaration and are identified with the help of the PB index. Any transition in M can be identified by the tuple (s_i, x, PB) where s_i is the start state of the transition, x is the input declaration and PB is the predicate branch taken. This algorithm also has complexity of $O(|T|)$ and can be included with the transition ranking and input declaration enumeration processes.

i	P^i	g_{P^i}	g_D	PB
AK	num	$num = number$	$number = 0$	1
AK	num	$num = number$	$number = 1$	2
AK	num	$num \neq number$	$countr \geq 4$	3
AK	num	$num \neq number$	$countr < 4$	4

Figure 4.6: Partial transition table for transitions starting from S_s in EFSM M with PBs

queue - a first-in, first-out (FIFO) data structure
A_{s_k} - adjacency list that stores the information of which states in M are one transition away from state s_k. A is built by the algorithm on Figure 4.9.
$A_{s_k,r}$ - the r-th transition initiating from state s_k
$|A_{s_k}|$ - the number of states in M that are one transition away from state s_k
L_{s_m,s_v} - a matrix representing the length of the shortest path from state s_m to s_v
n - the number of states in M

Figure 4.7: Variables for the ϕ function algorithm

Incorrect representation of the PB in the GA phenotype can create undesirable bias towards some transitions. Since there can be different number of transitions sharing the same start state and input declaration for each state in M the lowest common multiple (LCM) of the number of PB values in all the states can be used as a common PB index range. This value can be divided as appropriate for each state to evenly distribute all the possible transitions sharing the same start state and input declaration. For example consider an FSM where transitions that share the same input declaration and start state have 1 to 4 PBs. In order to represent the PB as a number in the genotype the LCM value of 12 can be used and divided as necessary for the start state and input declaration considered (i.e. $r/12$ for 1 PB, $r/6$ for 2, $r/4$ for 3 and $r/3$ for 4 PBs). If instead of using the LCM value we used a simple 1 to 4 representation then the cases where there are only 3 PBs will have to be treated as special cases that might create unwanted bias in the gene representation towards some values (i.e. 1 represents 1 PB, 2 represents 2, 3 represents 3 and 4 represents either 1,2 or 3 PBs).

The *function* ϕ is used in the FTP fitness function and the FTP verification algorithms. $\phi(s_i, s_j)$ returns the number of transitions state s_i is away from state s_j, which can be useful in guiding the search for an FTP. The complexity of this algorithm is of $O(n.max(n, |T|))$ and it generates an $n \times n$ path length matrix (ϕ matrix). Since we are considering initially connected EFSMs, $n \leq |T|$ the algorithm's

```
for($m := 1$ to $n$) //for all the states in $M$
    for($k := 1$ to $n$) //initialise shortest path matrix
        $L_{s_m,s_k} := 0$
    endFor
    $queue := null$ //initialise empty queue
    $i := 0$ //initialise path length counter
    $L_{s_m,s_m} := 0$ //self loop
    add $s_m$ to $queue$ //add initial state of paths to queue
    while($|queue| > 0$) //while queue not empty
        $s_k :=$ get from queue //get state from queue
        //enumerate all the outgoing states from $s_k$
        for($s_v := A_{s_k,1}$ to $A_{s_k,|A_{s_k}|}$)
            //$s_v$ not the initial state AND path to it not yet assigned
            if($s_v \neq s_m$ AND $L_{s_m,s_v} = 0$)
                //reset path length counter if shorter path is available
                if($L_{s_m,s_k} < i$) then $i := L_{s_m,s_k}$
                $L_{s_m,s_v} := i + 1$
                add $s_v$ to $queue$
            endIf
        endFor
        $i := i + 1$ //increment path length counter
    endWhile
endFor
return $L$
```

Figure 4.8: ϕ function

complexity is $O(n.|T|)$. It would be useful to execute it once and store the results for subsequent use.

The algorithm for producing the function ϕ, adapted from [Gibb 85] is shown on Figure 4.8. The parameters involved are as follows: *queue* is a first-in, first-out (FIFO) data structure; A_{s_k} is the adjacency list that stores the information of which states in M are one transition away from a given state s_k of M; $A_{s_k,r}$ denotes the ending state of the r-th transition initiating from state s_k; $|A_{s_k}|$ is the number of states in M that are one transition away from state s_k where adjacency list A is built by the algorithm in Figure 4.9; L_{s_m,s_v} is a matrix representing the length of the shortest path from state s_m to s_v; and n is the number of states in M. The parameters involved are also shown in Figure 4.7.

The adjacency list is stored once and then reused. An algorithm for generating A is shown on Figure 4.9 and has a complexity of $O(n.|T|)$ [Gibb 85]. For example consider the Inres EFSM M_1 shown on Figure 4.1. The adjacency algorithm on Figure 4.9 was used to produce an adjacency table presented on Figure 4.10.

ϕ is based on a breath-first search algorithm, that finds the lengths of the shortest paths from a given state s_m to all other states in M with complexity of $O(max(n, |T|))$ [Gibb 85]. It uses the *queue* to trace through all the reachable states from a given state s_m and constructs a shortest path length matrix. The *queue* ensures a breadth-first search of all transitions. When executed for all n states in M its complexity becomes of $O(n.|T|)$.

This adjacency list is in turn used to generate results using the *function* ϕ. A worked example of the ϕ function for M_1 is shown on Figure 4.11. The values of the variables used in the ϕ function are traced at two points in the algorithm. The first is just after line 8 of the algorithm on Figure 4.8 and the second is just after line 14. Every time the algorithm exits the while loop after line 22 this is illustrated on Figure 4.11 by a line. When the variables are traced after line 8 of the algorithm s_k and s_v

```
for($k := 1$ to $n$) //for all the states in $M$
    $c := 1$
    for($j := 1$ to $|T|$) //for all the transitions in $M$
        $t_j \in T$
        $s_s := \Pi_1(t_j)$ //get start state of transition $t_j$
        if($s_k = s_s$) //if $t_j$ starts from $s_k$
            $s_f := \Pi_5(t_j)$ //get end state of transition $t_j$
            add $s_f$ to ordered list $A_{s_k,c}$
            $c := c + 1$
        endIf
    endFor
endFor
return $A$
```

Figure 4.9: Adjacency list A specification algorithm

start state	end state	transitions
0	0	t_{11}
0	1	t_1
0	2	none
0	3	none
1	0	t_{12}, t_4
1	1	t_3
1	2	t_2
1	3	none
2	0	t_{13}
2	1	none
2	2	none
2	3	t_5
3	0	t_8, t_{10}, t_{14}
3	1	none
3	2	t_{61}, t_{62}
3	3	t_7, t_9

Figure 4.10: Generated adjacency list for the Inres protocol EFSM M_1 4.1. States are 0 indexed.

are not yet initialised for that execution of the loop, hence no values are shown on the table for those variable.

algorithm line	queue top	s_m	i	s_k	s_v	A_{s_k,s_v}	L_{s_m,s_v}	L_{s_m,s_k}
8	0	0	0					
14	null	0	0	0	1	1	0	0
14	null	0	1	1	2	1	0	1
14	null	0	2	2	3	1	0	2
8	1	1	0					
14	null	1	0	1	0	2	0	0
14	0	1	0	1	2	1	0	0
14	null	1	2	2	3	1	0	1
8	2	2	0					
14	null	2	0	2	0	1	0	0
14	0	2	0	2	3	1	0	0
14	3	2	1	0	1	1	0	1
8	3	3	0					
14	null	3	0	3	0	3	0	0
14	0	3	0	3	2	2	0	0
14	2	3	1	0	1	1	0	1

Figure 4.11: Generated shortest path lengths matrix (ϕ function)for the Inres protocol EFSM M_1. States are 0 indexed.

The resulting shortest path length matrix is presented on Figure 4.12. These algorithm execution results are shown in order to illustrate how the algorithms work.

4.4.2 FTP search problem representations

Two FTP search problem representations are presented. In order to compare them each one is used with all four FTP generation techniques. The search techniques include two slightly different GAs, a random generation algorithm and Dijkstra's shortest (cheapest) path algorithm. Breadth first search algorithm is also used to evaluate the effectiveness of the fitness function. Figure 4.13 gives an example for one particular TP from M_1 (t_2 followed by t_{12}). The example illustrates how the two

start state	end state	distance
0	0	0
0	1	1
0	2	2
0	3	3
1	0	1
1	1	0
1	2	1
1	3	2
2	0	1
2	1	2
2	2	0
2	3	1
3	0	1
3	1	2
3	2	1
3	3	0

Figure 4.12: Generated shortest path lengths matrix (ϕ function) for the Inres protocol EFSM M_1. States are 0 indexed.

FTP search problem representations compare. Representation 1 uses all 4 preprocesses while representation 2 needs only two of those preprocesses listed in Figure 4.5. A phenotype representing a potential FTP is simply a list of transition references. However representation 1 uses an abstracted representation that identifies a potential FTP by a sequence of input character and PB tuples.

Representation	1:PBs	2:transitions
Preprocesses	1,2,3 and 4	1 and 4
Phenotype	2;12;	2;12;
EFSM representation	input 1 + PB 2	transition 2
	input 3 + PB 4	transition 12
IS/AIS	input indexed 1 in declarations list + PB 2	input for transition 2
	input indexed 3 in declarations list + PB 4	input for transition 12

Figure 4.13: Representation outline of FTP generation representations 1 and 2

As the two different representations of the FTP generation problem make the approaches slightly different we also refer to them as different FTP methods.

4.4.3 Representation 1: Fitness function using PBs

The first FTP search problem representation represents a transition using input characters and PBs. An FTP is considered invalid when M is in state s and there is no corresponding transition from s that can be triggered by the next input character in the generated input sequence. If a transition from the TP triggered by an AIS is invalid the EFSM stays in the same state and the next transition in the sequence is considered. Hence the whole AIS under consideration can be evaluated by the fitness function even when invalid transitions have been attempted.

This FTP search problem representation has the following characteristics:

- It uses a fitness function that rewards transition sequences with higher ranked transitions and penalises invalid transitions.
- It produces a numerical value potentially showing how close an input sequence is to defining a valid FTP.
- The fitness function penalises AISs that do not end at the desired final state.
- The fitness value of the sequence incurs a penalty for each invalid transition.
- The GA is directed towards generating input sequences that contain mostly easy to execute transitions and hence more likely to be feasible transitions.
- The fitness function represents the search for an FTP sequence as a function minimisation problem so an AIS with a lower fitness value is considered to be more likely to form an FTP since it is made up of more highly ranked transitions.

```
$faults := 0$
$strengthValue := 0$
$S_k := S_{FTP}$
if ($l = 0$) then return -1 //error code for empty FTP
for($i := 1$ to $l$) //for all the transitions in the sequence
    $S_m := S_k$
    $S_k := \delta'(S_m, x_i, PB_i)$
    if ($S_k \neq \emptyset$)
        //There is such a transition
        $strengthValue := strengthValue + r_{S_m,x_i,PB_i}$
    endIf
    else
        $S_k := S_m$ //No such transition
        $faults := faults + 1$
    endElse
endFor
return $strengthValue + faults.penaltyValue1 + \phi(S_k, S_{FTP'}).penaltyValue2$
```

Figure 4.14: Representation 1: PB notation - FTP fitness algorithm

The fitness does not check if a particular transition path has been followed. It makes sure that the end state is as near as possible to the target end state and that as many inputs as possible from the AIS are valid and trigger transitions in the EFSM.

The algorithm for the fitness function proposed is presented in Figure 4.14. The parameters for this function and the FTP verification algorithm presented in Figure 4.15 are as follows: S_{FTP} is the start state of the TP; $S_{FTP'}$ is the intended end state of the TP; min and max are the value range for randomly generated input parameters; $parList$ is the array of input parameters; C_V represents the current values of all the variables in the internal variable set V of M; x is a single input declaration; l is the length of the input sequence; r is the set of transition rankings; $r_{s,x,PB}$ represents the rank of the transition initiating from state s with input declaration x and predicate branch PB; $penaltyValue1$ is the penalty constant or function that penalises the fitness when an unspecified transition is triggered; $penaltyValue2$ is the penalty constant or function that penalises the fitness when the TP does not end at the desired state; $\delta(s_i, x, parList, C_V) = s_j$ is the EFSM state transfer function; $\delta'(s_i, x, PB) = s_j$ is the EFSM abstracted state transfer function; and $\phi(s_i, s_j) = n$ is the function that returns the number of transitions s_i is away from s_j. The parameters involved are also shown on Figure 4.16.

4.4.4 FTP verification

In order to evaluate the results of the experiment it is useful be able to estimate the quality of a TP. Such a quality measure can estimated how easy is it to trigger a TP or generate an IS that would trigger it.

In PB notation an AIS is used to represent a potential FTP. After an AIS has been generated it has to be checked to determine whether it represents a valid TP since the GA fitness need not indicate this. The FTP verification algorithm estimates how easy it is to produce an IS that corresponds to this AIS. The algorithm will dynamically verify how easy it is to trigger the transitions that the fitness function estimated as easily triggered.

We wish to estimate how easy is it to produce an IS for a TP in order to estimate the quality of TPs produced. This is done by making 1000 attempts at randomly generating an IS for a TP and calculating the success ratio. The FTP verification algorithm is presented in Figure 4.15 and the parameters involved summarised in Figure 4.16. The AIS for every TP is transformed into an IS and the algorithm generates random input parameters for each input declaration in the IS and compares the transition triggered with that of the AIS, indexed by start state s_i, input declaration x and PB. If for all the transitions in the sequence the end state of the transitions triggered is the same as in the AIS then the TP is considered to be feasible for that instant and that particular IS. For each TP the algorithm returns a value of 0 when no valid IS have been found to trigger that TP (such TPs cannot be guaranteed to be feasible) and a value of up to 1000 shows that some or all attempts succeeded.

```
01 if ($n = 0$) then return $-888$ //Empty sequence
02 $success := 0$
03 for ($repeats := 1$ to $1000$)
04     $failed := false$
05     $S_m := S_{FTP}$
06     for ($i := 1$ to $l$)
07         $parList := randomGeneration(min, max)$
08         $S_k := \delta(S_m, x_i, parList)$
09         $S_j := \delta'(S_m, x_i, PB_i)$
10         $S_k := S_i$
11         if ($S_k \neq S_j$) //incorrect transition
12             $failed := true$
13         endIf
14         $S_m := S_k$ //move to next transition
15     endFor
16     if ($failed = false$) //If no incorrect transitions
17         $success := success + 1$
18     endIf
19 endFor
20 return $success$
```

Figure 4.15: FTP Verification Algorithm

For example a TP with quality factor 900/1000 is considered better than one with quality factor 450/1000.

The FTP verification algorithm in Figure 4.15 only checks if the correct states of a TP are followed. Alternatively instead of just comparing the end states of the TP generated using an IS and AIS the actual transitions that are executed can be compared. The above FTP verification algorithm can easily be modified to do this by changing the condition on line 11. This type of strict verification may be useful when we require a particular sequence of transitions to be executed. Consider the example in Figure 4.17. The two transitions t_1 and t_2 both share the same input declaration, start and end states. Imagine we have selected t_1 for our TP because it has a g_D $v' > 0$, $v \in V$ whilst t_2 has a g_D $v' = 0$. As discussed previously a predicate with $>$ is considered to be more easily satisfiable than one with $=$. However for this particular

S_{FTP} - start state of TP
$S_{FTP'}$ - intended end state of TP
min, max value range for randomly generated input parameters
$parList$ - array of input parameters
C_V - the current values of all the variables in the internal variable set V of the EFSM
x - single input declaration
l - length of the input sequence
r - set of transition rankings
$r_{s,x,PB}$ - The rank of the transition initiating from state s with input declaration x and predicate branch PB
$penaltyValue1$ - penalty constant or function that penalises the fitness when an unspecified transition is triggered
$penaltyValue2$ - penalty constant or function that penalises the fitness when the TP does not end at the desired state
$\delta(s_i, x, parList, C_V) = s_j$ - EFSM state transfer function
$\delta'(s_i, x, PB) = s_j$ - EFSM abstracted state transfer function
$\phi(s_i, s_j) = n$ - function that returns the number of transitions s_i is away from s_j

Figure 4.16: Variables for the FTP fitness and FTP verification algorithm using PB notation

EFSM $v' > 0$ can be the more difficult to satisfy condition. If we only care about the end states of the TP, this fact may not affect the result. Although it is because of t_1 that the particular TP through s_i and s_j has been selected, the fact that t_2 is actually executed instead does not affect the end state of the TP.

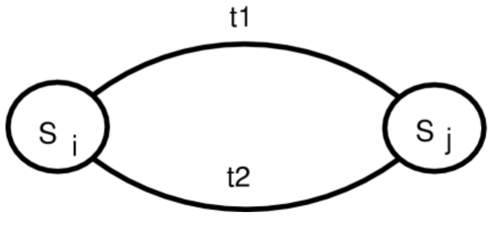


Figure 4.17: Example transitions within a TP

A TP could have properties other than just being feasible. It could for example identify or verify a state in the EFSM by generating a unique set of outputs. In this case it would be necessary to produce an IS that follows the exact TP of the AIS otherwise the unique properties of this path will be lost.

Section 4.5.2 describes the results of an empirical study of how FTP verification results compares to strict verification.

4.4.5 Representation 2: Fitness function using transition notation

An alternative and simpler representation of the search problem for FTP can be defined. Instead of representing a TP by means of AIS using PBs, a TP can be represented by the actual transitions that form it. All the transitions of an EFSM can be indexed and a genotype can represent a potential FTP by encoding a sequence of transition labels.

The algorithm for evaluating the fitness function used with this alternative FTP representation is presented on Figure 4.18. The parameters involved are as follows: S_{FTP} is the start state of the TP; $S_{FTP'}$ is the end state of the TP; T is the finite set of all transitions in M; T' is the transition sequence whose fitness is being determined; l is the length of the input sequence; r is the set of transition rankings; r_t is the rank for transition t; $penaltyValue1$ is the penalty constant or function that penalises the fitness when an unspecified transition is triggered or the transitions are not consecutive (i.e. not following each other); $penaltyValue2$ is the penalty constant or function that penalises the fitness when the TP does not end at the desired state; and $\phi(s_i, s_j) = n$ is the function that returns the number of transitions s_i is away from s_j. The parameters involved are also shown on Figure 4.19.

Before this fitness algorithm can be used a number of processes must be completed. These include two of the processes used by the FTP fitness algorithm for PB notation - transition ranking and ϕ. The general aim of the fitness is to correctly sequence transition labels so that they form a TP that is likely to be an FTP. Transitions are

```
$faults := 0$
$strengthValue := 0$
$S_k := S_{FTP}$
if ($l = 0$) then return -1 //error code for empty FTP
for($i := 1$ to $l$) //for all the transitions in $T'$
  $S_m := S_k$
  $S_k := \Pi_1(t_i)$ //Start state of transition $i$
  //if the transition initiates from the last reached state
  if($S_k = S_m$)
    $strengthValue := strengthValue + r_{t_i}$
  endIf
  //Ignore this transition since it does not initiate from $S_m$
  else
    $S_k := S_m$
    $faults := faults + 1$
  endElse
endFor
return $strengthValue + faults.penaltyValue1 + \phi(S_k, S_{FTP'}).penaltyValue2$
```

Figure 4.18: Representation 2: Transition notation - FTP fitness algorithm

ranked according the criteria described in Section 4.4.3. The function rewards higher ranked transitions and introduces a penalty when transitions are not sequential.

A transition from the transition sequence is considered invalid if it does not start from the state where the previous transition ended. If a transition is invalid the fitness value incurs a penalty and the next transition in the sequence is considered in turn by the algorithm. Hence the whole sequence of transitions under consideration can be evaluated by the fitness function even when invalid transitions have been attempted. The fitness function penalises TPs that do not end in the desired final state.

The fitness value of the sequence incurs a penalty for each invalid transition. Hence the GA is directed towards generating a sequence of transitions that contain mostly easy to execute transitions and hence more likely to be feasible transitions. The fitness function represents the search for an FTP sequence as a function minimisation problem so a sequence of transitions with a lower fitness value is considered to be

S_{FTP} - start state of TP
$S_{FTP'}$ - end state of TP
T - finite set of all transitions in M
T' - is the transition sequence where fitness is being determined
l - length of the input sequence
r - set of transition rankings
r_t - The rank for transition t
$penaltyValue1$ - penalty constant or function that penalises the fitness when an unspecified transition is triggered or the transitions are not consecutive (i.e. not following each other)
$penaltyValue2$ - penalty constant or function that penalises the fitness when the TP does not end at the desired state
$\phi(s_i, s_j) = n$ - function that returns the number of transitions s_i is away from s_j

Figure 4.19: Variables for the FTP fitness and FTP verification algorithm using transition notation

more likely to form an FTP since it is made up of more highly ranked transitions.

Once a transition sequence is generated it needs to be evaluated. An IS can be easily extracted from a sequence of transitions. In order to verify whether the IS can trigger an FTP it is possible to use a verification algorithm similar to the one described in Section 4.4.4. The verification algorithm for this FTP search problem representation is analogous to the verification algorithm on Figure 4.15. It returns an estimation of TP quality (representing feasibility) identical to the one defined in Section 4.4.4. The only slight difference is that the abstract state transfer function δ' is no longer necessary since we already know each transition in the sequence.

The verification function can generate a measurement value that represents the quality of a TP (as discussed in section 4.4.4) that can be used as a common feasibility measure. Hence results generated using the two different FTP search problem representations can be compared.

4.4.6 Generating FTPs using Dijkstra's algorithm

Dijkstra's algorithm is known to be an efficient algorithm to find the shortest path between vertices in weighted connected graphs [Dijk 59]. This algorithm can be applied to the problem of finding the shortest FTP between two states s_j and s_k. As discussed earlier, the feasibility of a TP cannot be guaranteed hence favouring certain transitions could guide the search towards finding a TP for which input sequence can easily be generated. The ranking methods explained earlier can be used here as well. Each transition in a TP can be given a weight corresponding to its rank. Then the algorithm will simply find the cheapest, in terms of weights, TP from s_j to s_k.

Once a TP is generated it needs to be evaluated in order to measure its quality (feasibility) and compare that value to the fitness function measure. The TP is verified and the quality factor calculated using the FTP verification algorithm shown in Figure 4.15.

Dijkstra's algorithm determines the distances (costs) between a given state s_j and all other states in an FSM. The algorithm begins at a specific state and extends outward within the FSM, until all states have been reached. Dijkstra's algorithm stores a summation of minimum cost transitions from s_j to all other states. We have extended the algorithm so it not only stores the cost of the cheapest path but also stores the actual path.

Dijkstra's algorithm creates labels associated with states. These labels represent the distance (cost) of s_j to a particular state s_k. Two kinds of labels are used, temporary and permanent. The temporary labels are given to states that have not been reached. The value given to these temporary labels can vary. Permanent labels are given to states that have been reached and their distance (cost) from s_j is known. The value given to these labels is the distance (cost) of s_j to that state. For any given state, there must be a permanent label or a temporary label, but not both.

Lets consider an example FSM M_1 on Figure 4.20. The algorithm begins by

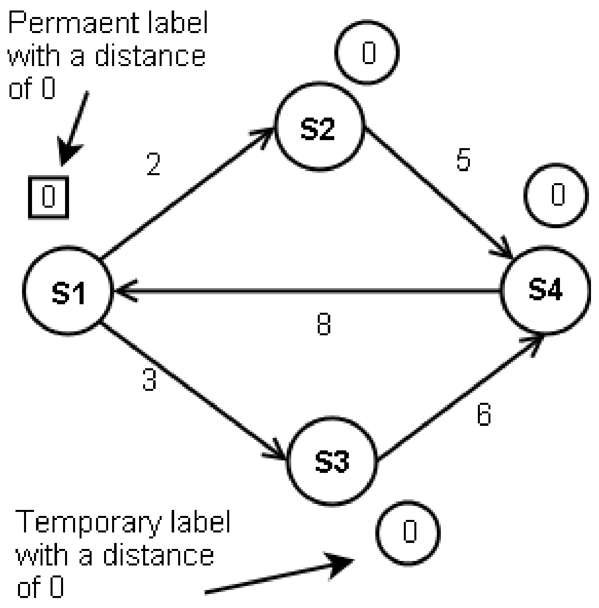


Figure 4.20: Dijkstra's algorithm example - step 1

initialising any state in the graph (s_1, for example) a permanent label (in square) with the value of 0, and all other states a temporary label (in circle) with the value of 0.

The algorithm then proceeds to select the least cost transition connecting a state with a permanent label (currently s_1) to a state with a temporary label (s_2, for example). The label of s_2 is then updated from a temporary to a permanent label. The value of the label for s_2 is then determined by the addition of the cost of the transition from s_1. Where there are multiple transitions between two states the least cost transition is selected (Figure 4.21).

The next step is to find the next least cost transition extending to a state with a temporary label from either s_1 or s_2 (s_3, for example), change the label of s_3 to permanent, and determine its distance from s_1 (Figure 4.22).

This process is repeated until the labels of all states in the FSM are permanent (Figure 4.23).

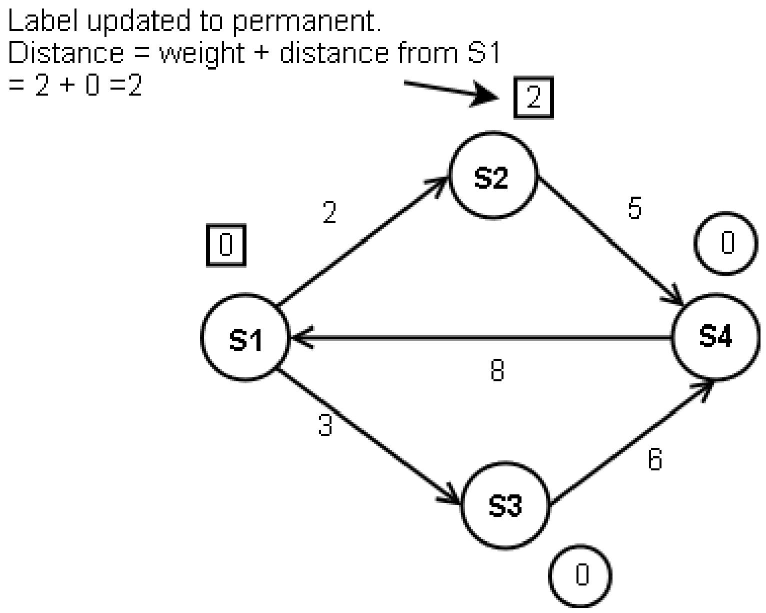

Label updated to permanent.
Distance = weight + distance from S1
= 2 + 0 =2
2
S2
2
5
0
0
S1
S4
8
3
6
S3
0

Figure 4.21: Dijkstra's algorithm example - step 2

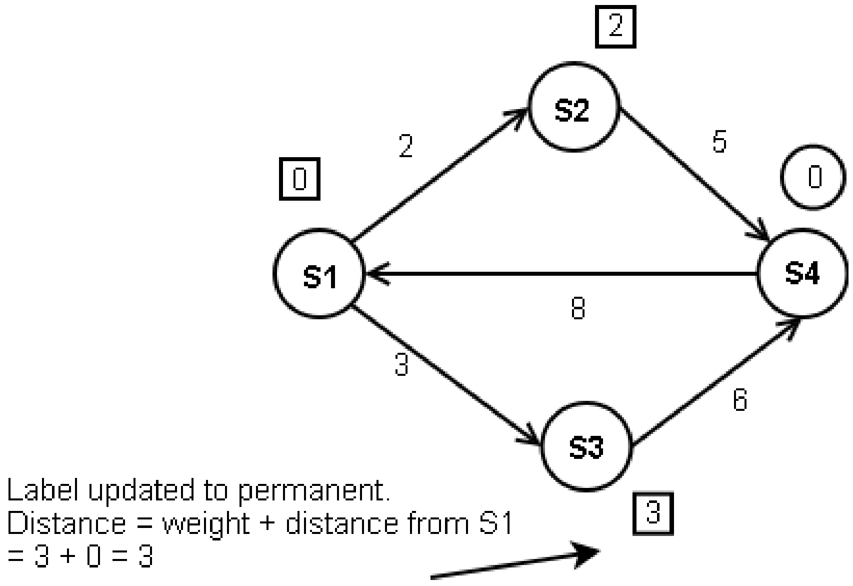

2
S2
2
5
0
0
S1
S4
8
3
6
S3
Label updated to permanent.
Distance = weight + distance from S1
= 3 + 0 = 3
3

Figure 4.22: Dijkstra's algorithm example - step 3

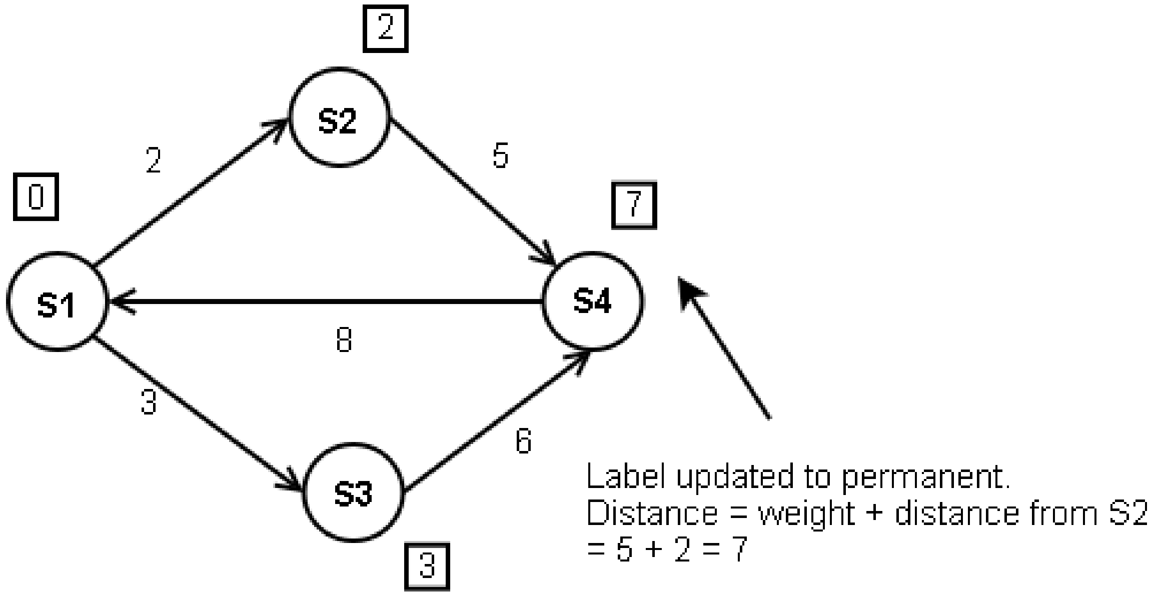

2
0
7
3
S1
S2
S3
S4
2
5
8
3
6
Label updated to permanent.
Distance = weight + distance from S2
= 5 + 2 = 7

Figure 4.23: Dijkstra's algorithm example - step 4

The adapted Dijkstra's algorithm we used is outlined in Figure 4.24. The parameters involved are as follows: S is the finite set of states in the FSM M; S' is the set of all states with permanent labels; s_1 is the source state; T is the finite set of transitions in M; n is the number of states in M; D is the set of distances (costs) from s_1; and P is the set of paths from s_1. The variables for this algorithm are summarised on Figure 4.25.

4.4.7 Advantages of heuristic search over Dijkstra's algorithm

The GA fitness algorithms uses static analysis of the EFSM M under consideration to rank transitions according to how easily they can be triggered. Some of the characteristics of these transitions however cannot be measured in this way.

For example consider two transitions t and t' in a sequence of a TP where the update function of t changes the value of internal variable $v := f(x)$ and one of the

```
$S' = \{s_1\}$
For ($i := 2$ to $n$)
    $D_i := T_{s_1,s_i}$
EndFor
For ($i := 2$ to $n$)
    choose a state $s_w \in (S - S')$ such that $D_w$ is minimum
    add $s_w$ to $S'$
    For each state $s_v \in (S - S')$
        $D_v := min(D_v, D_w + T_{s_w,s_v})$
        save minimal cost path to $P_v$
    EndFor
EndFor
```

Figure 4.24: Adapted Dijkstra's shortest (cheapest) path algorithm for an FSM

S	- finite set of states in M
S'	- finite set of all states with permanent labels
s_1	- initial state
T	- finite set of transitions in M
n	- number of states in M
D	- set of distances from s_1
P	- set of paths from s_1

Figure 4.25: Variables for the adapted Dijkstra's shortest (cheapest) path algorithm

guards in t' uses the value of v. We might desire the fitness function to avoid such transition pairs as they could lead to the generation of TPs such that generating input sequences to trigger them is more difficult. Other similar analysis of the TP can be added to the fitness functions in an attempt to direct the search better.

Dijkstra's shortest path algorithm can efficiently be used in path search problems where all the weighting (ranking) of the transitions are fixed. It is not applicable when the weighting of a transition depends on the rest of the path. When it comes to the dynamic (run-time) analysis the problem changes. Using a fitness algorithm and a heuristic search allows for much more flexible search that can incorporate not only static but also dynamic analysis.

This work on generating FTPs has application beyond EFSM IS generation. It is the first step in an attempt to resolve problems related to generating test ISs for EFSMs. The reason for using heuristic search for a problem that can be solved using an efficient shortest-path algorithm is that we are developing the heuristic search that can be used for related IS generation problems for which Dijkstra's shortest-path algorithm is not appropriate (Chapter 5).

4.5 Experiments

In this section we outline a set of experiments and present the results. A breadth first search (BFS) algorithm is first used to generate a set of FTPs for two EFSMs. The effectiveness of the fitness function to estimate feasibility is evaluated using these results. Next the fitness function is used to guide heuristic search for FTPs using the two different FTP search problem representations previously described in this chapter.

4.5.1 Experiment strategy

The aim of the experiments is to evaluate the effectiveness of the fitness function defined and compare different approaches for generating FTPs. First BFS algorithm is used to generate all FTPs within a set range for a set of EFSMs. The two different FTP search problem representations are then used with the GA and Random generation algorithms and compared. We propose two slightly different search techniques for generating FTPs using GAs. For comparison the amount of effort used by the GAs is given to a random generation algorithm and this uses the same FTP verification method as the GAs. The BFS algorithm results are used to evaluate the effectiveness of the fitness function to estimate feasibility.

The aim is to find an FTP between two states (BF-FTP) of an EFSM such that it is easy to generate an input sequence that triggers it. Hence every potential FTP produced has its quality factor estimated (as explained in Section 4.4.4). The four FTP generation techniques were discussed in Section 4.4.

FTP generation using Dijkstra's algorithm is straightforward. Dijkstra's algorithm finds the shortest paths using weighted edges and these paths can be used as potential FTPs. If the EFSM edges are weighted using the same criteria used by the GA fitness algorithms then Dijkstra's algorithm will always generate the optimal results.

BFS algorithm is used to generate all possible TPs within a specified length. The fitness and FTP quality is calculated for each TP. These results will include both the optimal TPs that Dijkstra's algorithm can find and TPs that cannot be guaranteed to be FTPs. Hence these results could be an indication of how successful the fitness function is in directing the search for FTPs. Comparing the results of the GAs with that of the random generation algorithm on the other hand could indicate whether using the chosen fitness function and representation of the search problem give an advantage to heuristics like GAs over random generation.

Although Dijkstra's shortest path algorithm might seem the obvious choice we

n - The number of states in the given EFSM
att - The number of attempts to generate an FTP sequence with a specified length for each pair of states in the EFSM
min - The shortest FTP to be generated
max - The longest FTP to be generated
l - FTP length being generated
i - the attempt number
s_1 - the initial state of an EFSM

Figure 4.26: Variables for the test strategy algorithm

```
for($s' := 1$ to $n$) //for all states in $M$
    for($l := min$ to $max$) //for all the TP lengths
        for($i := 1$ to $att$) //for all the repeated attempts
            $FTP_{s_1,s'}$ := attempt to generate FTP with length $l$
            //Optional exit after the shortest FTP is found
            if($FTP_{s,s'}$ valid FTP from $s$ to $s'$)
                $i := att$ //exit loop and move to next length
            endIf
        endFor
    endFor
endFor
```

Figure 4.27: Test strategy algorithm

have to remember that the problem of generating an FTP is just the first part of the more difficult problem of generating actual test sequences that do not always represent the shortest path between two states. Also there are some other computationally inexpensive analysis of a potential FTP that can be added to the existing GA fitness functions to make it direct the search for FTPs more accurately (Section 4.4.7). Such information cannot always be represented using a fixed weight for each edge, and hence Dijkstra's algorithm will not always be appropriate.

A set of experiments was conducted on a set of EFSMs. For each EFSM M a strategy is used that attempts to generate FTPs between the initial state of the EFSM and all other states.

By using M's initial state for all FTPs M would be in the same initial configuration (start state and values of the internal variables) with a simple reset. The problem of placing M in a given configuration before input sequence execution, other than its initial configuration, is not a focus of this work. A reset is not a necessary condition for the FTP generation but we assume a reliable reset for this test strategy in order to do multiple executions of M without considering the problems associated with placing M in a given configuration.

A test strategy attempting to generate the lowest cost FTP between the initial state of M and all states in M is shown on Figure 4.27. The parameters involved are as follows: n is the number of states in the given EFSM; att is the number of attempts to generate an FTP sequence with a specified length for each pair of states in the EFSM; min is the shortest FTP to be generated; max is the longest FTP to be generated; l is the FTP length being generated; i is the attempt number; and s_1 is the initial state of the EFSM. The parameters involved are also shown on Figure 4.26. The algorithm attempts to generate an FTP between the initial state s_1 of M and every state in M (including s_1). No more than att number of attempts are made for every FTP length ranging from min to max. In order to generate comparable results, given an EFSM the att, min and max attributes are kept the same for the different heuristic FTP generation technique - GAs and Random. For each different FTP generation technique the same test strategy is used but the appropriate FTP search problem representation used (GA with PB notation, GA with transition notation and random generation). FTP generation using Dijkstra's algorithm does not use the test strategy in Figure 4.27 as the algorithm finds the shortest path between a given state s and all states in M. The BFS algorithm also does not use the test strategy. It finds all paths of specified length between a given state s and all states in M.

4.5.2 Fitness evaluation results

This section describes results of a limited empirical study of how effective the proposed fitness algorithm is in estimating the feasibility of a TP in an EFSM. The study is limited to two EFSMs (Inres protocol M_1 and a Class 2 transport protocol M_2). TPs of length 1 to l, initiating from the initial state of M_1 and the initial state of M_2 were generated using BFS algorithm. The TPs were generated using a BFS algorithm. The TP generation was irrespective of notation since transition and predicate branch notations represent the same TPs. For ease of presentation the TP examples are shown in transition notation.

The fitness function attempts to estimate how easily a TP can be executed (hence how easily can the input to trigger this TP be generated). The evaluation of the quality factor of an FTP involved 1000 attempts to execute that TP with random input parameters. A negative statistical correlation is expected between the fitness values and the TP quality factor values. Correlation factor below 0.4 generally indicates lack of correlation while correlation factor of 0.6 and above indicates fair correlation between two sets of values considered [Pear 79]. Correlation factor between 0.4 and 0.6 is considered as some correlation. These limits vary for applications in different fields.

Class 2 Transport Protocol

The Class 2 transport protocol M_2 is presented in Figure 4.28 and the corresponding transition table is shown in Table 4.29 and Table 4.30. M_2 has more states, transitions and is more complex than M_1. M_2 is a major module (based on the AP-module [Boch 90]) of a simplified version of a class 2 transport protocol [Rama 03]. M_2 represents only the core transitions of that EFSM, as used in [Rama 03].

1083 TPs were generated and plotted in Figure 4.31 that represent all possible

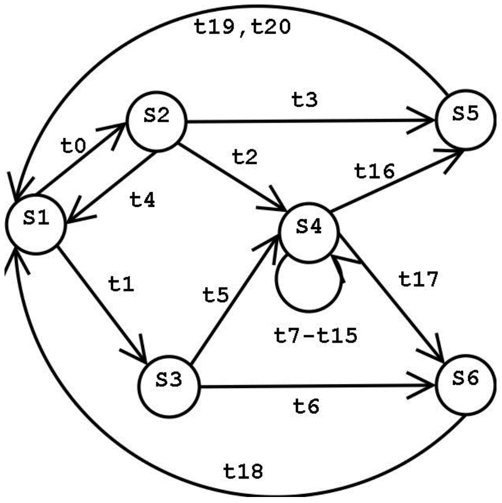


Figure 4.28: Class 2 transport protocol EFSM M_2

t	s_{start}	s_{end}	i	Output	Ranking
t_0	s_1	s_2	U?TCONreq	N!TrCR	0
t_1	s_1	s_3	N?TrCR	U!TCONind	0
t_2	s_2	s_4	N?TrCC	U!TCONconf	3
t_3	s_2	s_5	N?TrCC	U!TDISind N!TrDR	4
t_4	s_2	s_1	N?TrDR	U!TDISind N!terminated	0
t_5	s_3	s_4	U?TCONresp	N!TrCC	1
t_6	s_3	s_6	U?TDISreq	N!TrDR	0
t_7	s_4	s_4	U?TDATAreq	N!TrDT	2
t_8	s_4	s_4	N?TrDT	U!DATAind N!TrAK	3
t_9	s_4	s_4	N?TrDT	U!error N!error	3
t_{10}	s_4	s_4	U?U_READY	N!TrAK	0
t_{11}	s_4	s_4	N?TrAK		6
t_{12}	s_4	s_4	N?TrAK	U!error N!error	4
t_{13}	s_4	s_4	N?TrAK		7
t_{14}	s_4	s_4	N?TrAK	U!error N!error	5
t_{15}	s_4	s_4	N?Ready	U!Ready	2
t_{16}	s_4	s_5	U?TDISreq	N!TrDR	0
t_{17}	s_4	s_6	N?TrDR	U!TDISind N!TrDC	0
t_{18}	s_6	s_0	N?terminated	U!TDISconf	0
t_{19}	s_5	s_0	N?TrDC	N!terminated U!TDISconf	0
t_{20}	s_5	s_0	N?TrDR	N!terminated	0

Figure 4.29: Transition table for M_2 excluding transition guards

t	g_{P_i} and g_D	Ranking
t_0		0
t_1		0
t_2	$opt\_ind \leq opt$	3
t_3	$opt\_ind > opt$	4
t_4		0
t_5	$accpt\_ind \leq opt$	1
t_6		0
t_7	$S\_credit > 0$	2
t_8	$R\_credit \neq 0 \wedge Send\_sq = TRsq$	3
t_9	$R\_credit = 0 \wedge Send\_sq \neq TRsq$	3
t_{10}		0
t_{11}	$TSsq \geq XpSsq \wedge$ $cr + XpSsq - TSsq \geq 0 \wedge cr + XpSsq - TSsq \leq 15$	6
t_{12}	$TSsq \geq XpSsq \wedge$ $(cr + XpSsq - TSsq < 0 \vee cr + XpSsq - TSsq > 15)$	4
t_{13}	$TSsq < XpSsq \wedge$ $cr + XpSsq - TSsq - 128 \geq 0 \wedge cr + XpSsq - TSsq - 128 \leq 15$	7
t_{14}	$TSsq < XpSsq \wedge$ $(cr + XpSsq - TSsq - 128 < 0 \vee cr + XpSsq - TSsq - 128 > 15)$	5
t_{15}	$S\_credit > 0$	2
t_{16}		0
t_{17}		0
t_{18}		0
t_{19}		0
t_{20}		0

Figure 4.30: Transition guards for M_2

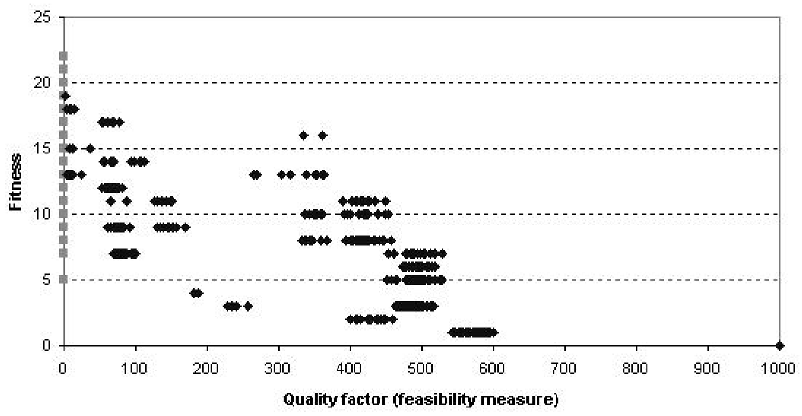


Figure 4.31: All TPs generated using BFS algorithm for M_2 with 1-5 transitions (correlation factor 0.72, when excluding the 0 quality factor TPs the correlation factor is 0.76). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

TPs with length of up to 5 transitions. 479 of these TPs have positive FTP quality factor (i.e. every TP was successfully triggered at least once in 1000 attempts). All the 604 TPs with 0 quality factor are drawn in Figure 4.31 using lightly shaded squares. As explained in Section 4.4.4 TPs with 0 quality factor cannot be guaranteed to be feasible (according to our feasibility estimation). Although the fitness aims to estimate feasibility a high proportion of TPs did not seem feasible for this EFSM. However all such TPs with 0 quality factor have a comparatively high fitness value.

This high proportion of TPs with 0 quality factor indicates that some initial TP conflict resolution could be used to improve the results. Conflict resolution for EFSM test sequences has been addressed in [Dual 04] and the use of such methods remains a topic for future work.

Some interesting FTPs from Figure 4.31 and their respective fitness values and quality factors are listed in Table 4.32.

The minimising fitness function function correctly identified all TPs that were made of only *simple* transitions. Such TP had a fitness value of 0 and a quality factor

	TP	fitness	quality factor (feasibility)
1	1;6;18;0;4;	0	1000/1000
2	1;5;16;20;1;	1	603/1000
3	0;2;7;	3	481/1000
4	0;2;11;11;11;	19	2/1000
5	1;5;14;11;11;	18	5/1000
6	1;5;15;14;11;	14	50/1000
7	0;2;10;11;10;	7	73/1000
8	0;4;0;2;11;	7	83/1000
9	1;5;10;15;11;	9	65/1000
10	0;2;10;14;16;	6	473/1000
11	0;2;10;11;16;	7	69/1000
12	0;2;14;14;14;	16	337/1000
13	0;2;7;14;14;	13	240/1000
14	1;5;14;14;10;	11	436/1000

Figure 4.32: Example TPs for the Class 2 transport protocol in transition notation.

1000/1000 (e.g. TP 1 in Table 4.32). TPs that contain mostly *simple* transitions but also a few conditional transitions have fitness value slightly above 0 and corresponding quality factor of about 500/1000 (e.g. TP 2 and 3 in Table 4.32). A disproportionate drop in the TP quality factor as soon as conditional transitions are considered is observed on Figure 4.31. The introduction of a single conditional transition to a TP can generally be expected in theory (all other things being equal) to halve the quality factor. Every additional condition in theory should further reduce the remaining quality factor. This could explain the two clusters of points between the 0-200/1000 and 400-600/1000 quality factor values.

Most TPs with quality factor below 200/1000 have high fitness values. In those cases the fitness algorithms has correctly estimated these TPs as not easy to execute. TPs 4,5 and 6 in Table 4.32 are such examples.

There are some paths that the fitness algorithm estimated as not too hard to execute (i.e. with relatively low fitness value) that have surprisingly low quality factor values (e.g. TPs 7,8 and 9 in Table 4.32). When all of the transitions involved in those TPs are examined in more detail $t11$ of M_2 appears to be in all those TPs. To illustrate the effect of this transition in a TP path consider TPs 10 and 11 in Table 4.32. The only difference between them is that the fourth transition of TP 11 is $t11$. Their fitness values differs only by one point, indicating that $t11$ is estimated to be more difficult to execute than $t14$ (used in TP 10 instead of $t11$). However the sharp contrast in the quality factor values indicates that $t11$ is harder to execute than the fitness function has estimated for this transition sequence. A more detailed analysis of the predicates before they are ranked, as previously discussed in this chapter, could help prevent this. This development remains future work.

There were some TPs with high fitness values (i.e. estimated to be hard to execute) that have relatively high feasibility ratio. Such examples included TPs 12, 13 and 14 in Table 4.32. All TPs with such characteristics seem to contain $t14$. Here the fitness function has not discovered that even though $t14$ has a number of complex conditions, they are easy to satisfy for this transition sequence. Similar to the analogous example above a more complex analysis of the transitions could consider such factors in the transition ranking and fitness generation and attempt to more accurately estimate the feasibility of TPs that include transitions like $t11$ and $t14$.

The fitness function seems to correctly estimate the feasibility of most of the 1083 TPs. There is a negative correlation factor of 0.72 between the fitness function and the quality factor illustrated on Figure 4.31. If we only consider the 479 FTPs the correlation factor is 0.76. The FTP results (including unsuccessful TPs) are classified in six sets according to the end state of the FTPs. The results for s_1, s_2, s_3, s_4, s_5 and s_6 are presented in Figures 4.33, 4.34, 4.35, 4.36, 4.37 and 4.38 accordingly. The cumulative results have negative correlation factors of 0.83, 0.95, 0.95, 0.69, 0.71 and 0.76 for TPs of s_1, s_2, s_3, s_4, s_5 and s_6 accordingly and negative correlation factors of 0.77, 0.87, 0.87, 0.75, 0.70 and 0.78 for the 479 FTPs accordingly.

This relatively strong correlation shows that even though the fitness function

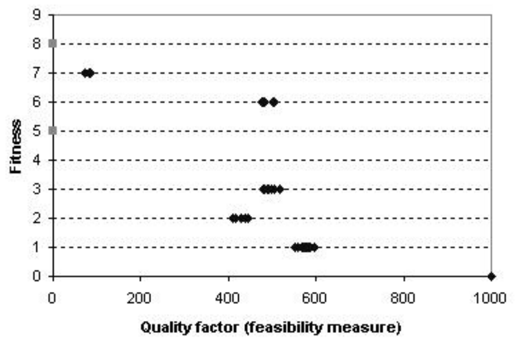


Figure 4.33: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_1 (correlation factor 0.83, when excluding the 0 quality factor TPs the correlation factor is 0.77). Light shaded squares represent unfeasible TPs. The dark diamonds represent FTPs.

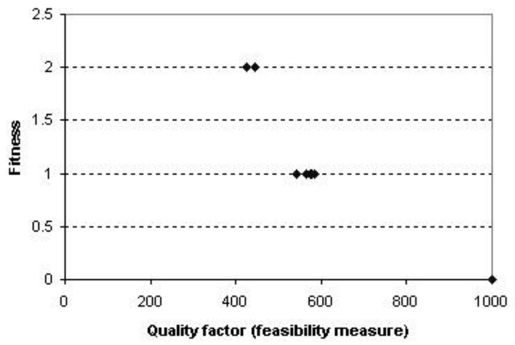


Figure 4.34: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_2 (correlation factor 0.95). The dark diamonds represent FTPs. No zero-quality TPs were found.

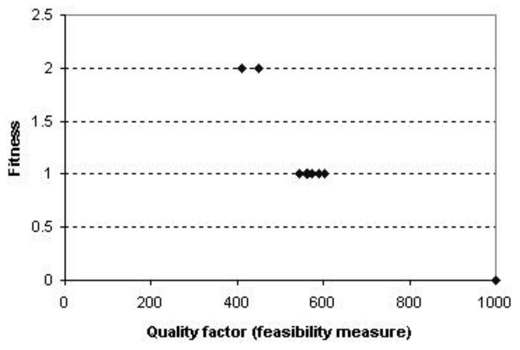


Figure 4.35: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_3 (correlation factor 0.69, when excluding the 0 quality factor TPs the correlation factor is 0.76). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

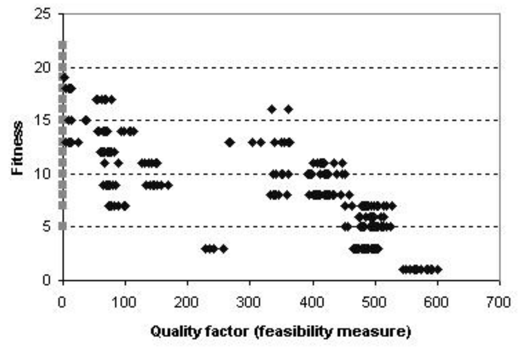


Figure 4.36: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_4 (correlation factor 0.69, when excluding the 0 quality factor TPs the correlation factor is 0.75). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

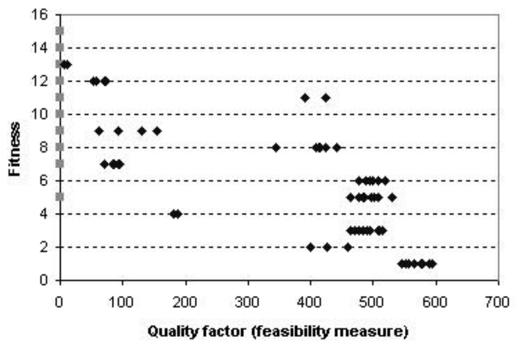


Figure 4.37: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_5 (correlation factor 0.71, when excluding the 0 quality factor TPs the correlation factor is 0.70). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

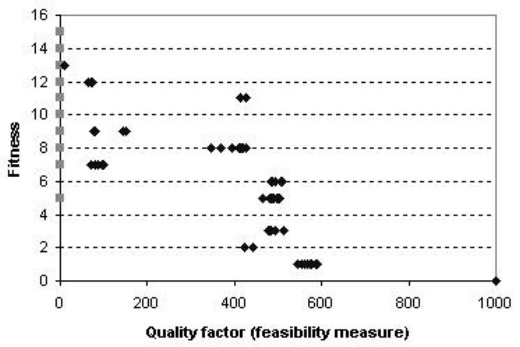


Figure 4.38: All FTPs generated using BFS algorithm for M_2 with 1-5 transitions ending at s_6 (correlation factor 0.76, when excluding the 0 quality factor TPs the correlation factor is 0.78). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

underestimated TPs with $t11$ and overestimated TPs with $t14$, it can reasonably estimate the feasibility of a TP without executing it.

The results presented use strict FTP verification (the exact TP has to be followed). When the same experiment was done using the relaxed FTP verification where only the start and end states of the TP are of importance (Figure 4.15) the results generated a negative correlation factor of only 0.13 and when only considering FTPs -0.22. This suggests that the reasons for using the relaxed FTP verification outlined in Section 4.4.4 might be EFSM dependant. It shows that the fitness is better at assessing feasibility of a given path than whether the corresponding IS can take us to a particular state. Although the relaxed FTP verification might be useful in cases like the one shown on Figure 4.17, strict FTP verification seems to perform much better (at least for this EFSM).

Inres Protocol

The EFSM M_1 in Figure 4.1 representing the Inres protocol that is simpler than the Class 2 transport protocol M_2 previously examined. The transition table for M_1 is presented in Figure 4.39.

The BFS algorithms was used to generate 257 TPs for M_1 where only 7 were not FTPs. Figure 4.40 illustrates those TPs that represent all possible TPs with length of up to 6 transitions with positive quality factor (i.e. every TP was successful triggered successfully at least once in 1000 attempts). Again TPs with quality factor of 0 cannot be guaranteed to be feasible and drawn with a lighter shaded squares. Here however there are considerably less TPs with 0 quality factor for M_1. This could indicate that the FTPs are easier to generate.

Some interesting FTPs from Figure 4.40 and their respective fitness values and quality factors are listed in Figure 4.41.

The minimising fitness function correctly identified all TPs that were made of

t	s_{start}	s_{end}	i	Output	g_{P_i} and g_D	Ranking
t_0	s_d	s_w	ICONreq	!CR		0
t_1	s_w	s_c	CC	!ICONconf		0
t_2	s_w	s_w	T_expired	!CR	$counter < 4$	2
t_3	s_w	s_d	T_expired	!IDISind	$counter \geq 4$	1
t_4	s_c	s_s	IDATreq	DT		0
t_5	s_s	s_c	AK		$num = number \wedge number = 0$	6
t_6	s_s	s_c	AK		$num = number \wedge number = 1$	6
t_7	s_s	s_s	AK	DT	$num \neq number \wedge couter < 4$	5
t_8	s_s	s_d	AK	!IDSind	$num \neq number \wedge couter \geq 4$	4
t_9	s_s	s_s	T_expired	DT	$counter < 4$	3
t_{10}	s_s	s_d	T_expired	!IDSind	$counter \geq 4$	2
t_{11}	s_d	s_d	DR	!IDSind		0
t_{12}	s_w	s_d	DR	!IDSind		0
t_{13}	s_c	s_d	DR	!IDSind		0
t_{14}	s_s	s_d	DR	!IDSind		0

Figure 4.39: Transition table for the Inres protocol on Figure 4.1.

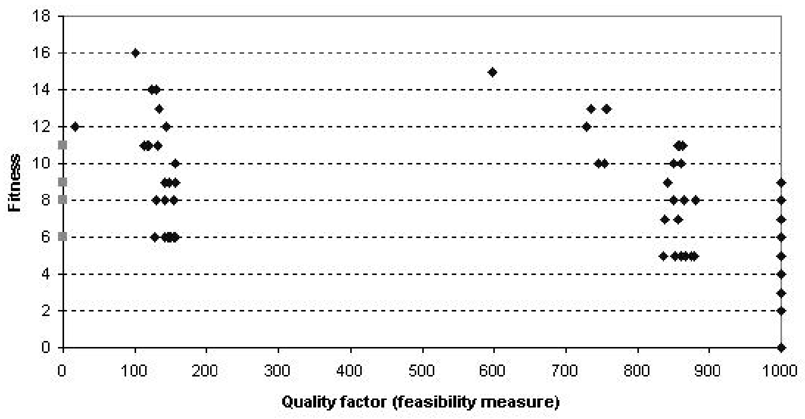


Figure 4.40: All FTPs generated using BFS algorithm for M_1 with 1-6 transitions (correlation factor 0.62, when excluding the 0 quality factor TPs the correlation factor is 0.60). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

	TP	fitness	quality factor (feasibility)
1	0;1;4;14;	0	1000/1000
2	20;20;0;4;12;	0	1000/1000
3	0;1;4;7;7;6;	16	101/1000
4	0;2;1;4;7;6;	13	134/1000
5	0;1;4;9;7;6;	12	144/1000
6	11;11;0;1;4;6;	6	127/1000
7	0;1;4;6;4;14;	6	142/1000
8	11;11;0;4;7;	5	860/1000
9	11;11;0;4;14;	0	1000/1000
10	0;1;4;6;4;	6	149/1000
11	0;1;4;6;4;5;	12	16/1000
12	0;1;4;9;9;9;	9	1000/1000
13	0;2;2;2;2;3;	9	1000/1000
14	0;2;1;4;9;7;	10	860/1000
15	0;1;4;7;7;	10	745/1000

Figure 4.41: Example TPs for the Inres protocol in transition notation.

only *simple* transitions just as it did for M_2. Such TPs had a fitness value of 0 and a quality factor 1000/1000 (e.g. TP 1,2 and 9 in Figure 4.41). TPs containing mostly *simple* transitions tend to have a high quality factor and low fitness value (e.g. TP 8 in Figure 4.41 has quality factor 860/1000 and fitness value of 5). Different to the results for M_2 the introduction of guarded transitions in the TPs generated for M_1 does not result in a halving of the quality factor. Even though a condition was introduced in the TP, the TP quality factor seem to remain high. This could indicate that the introduced condition is easy to satisfy. Here the fitness function seems to be penalising t_7, the only conditional transition in TP 8 slightly more than necessary. To illustrate the effect of this transition in a TP path consider TPs 8 and 9 in Figure 4.41. The only difference between them is that the sixth transition of TP 8 is t_7. Their fitness values differ by 5 points but they both have high quality factors. This indicates that t_7 is estimated to be much more difficult to execute than t_{14} (used in TP 9 instead of t_7). However the sharp contrast in the fitness values indicates that t_7 is slightly easier to execute than the fitness function has estimated for this transition sequence. A more detailed analysis of the predicates before they are ranked, as previously discussed in this chapter, could help prevent this.

A large number of TPs with quality factor below 200/1000 have high fitness values. In those cases the fitness algorithms has correctly estimated these TPs as not easy to execute. TPs 3, 4 and 11 in Figure 4.41 are such examples.

There as some paths that the fitness algorithm estimated as not too hard to execute (i.e. with relatively low fitness value) that have surprisingly low quality factor values (e.g. TPs 6 and 7 in Figure 4.41). When all of the transitions involved in those TPs are examined in more detail t_6 of M_1 appears to be in all those TPs. To illustrate the effect of this and a similar transition t_5 in a TP path consider TPs 10 and 11 in Figure 4.41. The only difference between them is that the sixth transition of TP 11 is t_5. t_6 is the only conditional transition in TP 10 but the fitness value is only 6. This suggests that executing t_6 is more difficult than the fitness function estimated. Similarly t_5 seems to have similar properties as TP 11 has a quality factor of only 16/1000. A more detailed analysis of the predicates before they are ranked could be useful in this example as well.

There were some TPs with high fitness values (i.e. estimated to be hard to execute) that have considerably high quality factor. Such examples included TPs 12, 13, 14 and 15 in Figure 4.41. All TPs with such characteristics seem to contain t_2, t_7 or t_9. We already discussed that t_7 seems to be penalised too harshly. Consider t_2 and t_3 in Figure 4.39. t_3 is estimated to be easier to execute than t_2 due to the difference in the comparison operators of their g_D. However when the EFSM is closely examined t_2 appears to always be feasible for four consecutive executions and only then t_3 becomes feasible for a single execution. t_2 represents a counter loop and t_3 is the loop exit. After four consecutive executions of t_2, it becomes infeasible and the counter exit t_3 is executed. Similarly t_9 represents a counter loop and t_{10} is the loop exit. The dynamic behaviour of these transitions is difficult to estimate using a simple ranking process. So, identification of such loops could be beneficial. Detailed analysis of the predicates involved in this example could have strongly benefited the fitness function.

The fitness function seems to correctly estimate the feasibility of most of the 257 TPs. There is a negative correlation factor of 0.62 between the fitness function and the quality factor illustrated in Figure 4.40. If we only consider the 479 FTPs the correlation factor is 0.6. The FTP results are classified in four sets according to the end state of the FTPs. The results for s_d, s_w, s_c and s_s are presented in Figures 4.42, 4.43, 4.44 and 4.45 accordingly. The cumulative results have negative correlation factors of 0.62, 0.54, 0.85 and 0.49 for FTPs of s_d, s_w, s_c and s_s accordingly. The correlation factor seems to be lowest for the two states s_w and s_s, target states for t_2 and t_9 accordingly and negative correlation factors of 0.61, 0.5, 0.87 and 0.54 for the 479 FTPs accordingly.

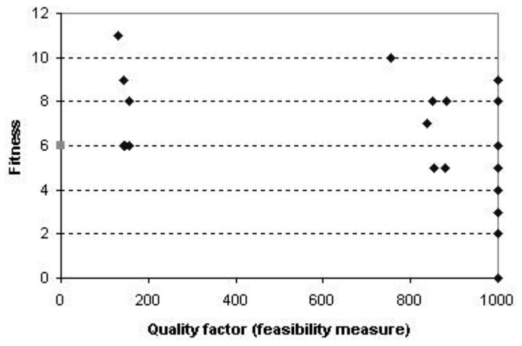


Figure 4.42: All FTPs generated using BFS algorithm for M_1 with 1-6 transitions ending at s_d (correlation factor 0.62, when excluding the 0 quality factor TPs the correlation factor is 0.61). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

The correlation between the fitness function and the TP quality factor is not as strong as that for M_2 however it still seems to correctly estimate the feasibility for much more than half the FTPs. The small size of the EFSM and the complex

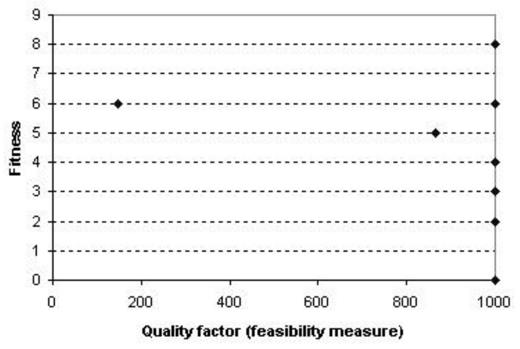


Figure 4.43: All FTPs generated using BFS algorithm for M_1 with 1-6 transitions ending at s_w (correlation factor 0.54, when excluding the 0 quality factor TPs the correlation factor is 0.5). The dark diamonds represent FTPs. No zero-quality TPs were found.

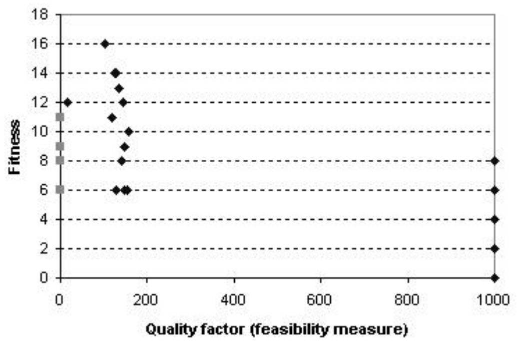


Figure 4.44: All FTPs generated using BFS algorithm for M_1 with 1-6 transitions ending at s_c (correlation factor 0.85, when excluding the 0 quality factor TPs the correlation factor is 0.87). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

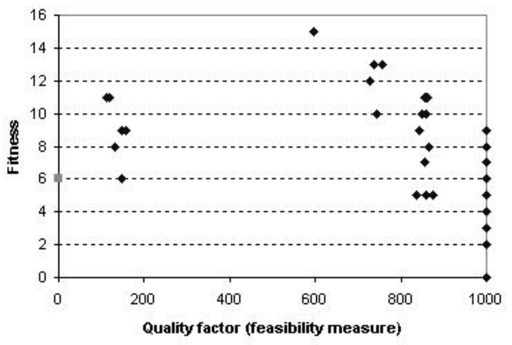


Figure 4.45: All FTPs generated using BFS algorithm for M_1 with 1-6 transitions ending at s_s (correlation factor 0.49, when excluding the 0 quality factor TPs the correlation factor is 0.54). Light shaded squares represent zero-quality TPs. The dark diamonds represent FTPs.

dynamic behaviour of some of the transitions (loops of fixed number of iterations) seem to contribute towards this result.

The results presented use strict FTP verification (the TP has to be obeyed). Initially similar results were generated using the relaxed FTP verification where only the start and end states of the FTP are of importance. Those results generated a negative correlation factors of 0.62 for all TPs and 0.61 for FTPs. Again the small size of the EFSM is the likely cause for similar correlation ratio values.

4.5.3 FTP generation results

Following the positive results in evaluating the fitness function, the test strategy in Figure 4.27 is used to generate FTPs. Both FTP search problem representations are employed to generate FTPs using the two GA and a random generation search techniques outlined earlier. The results are compared and conclusions drawn. The test strategy in Figure 4.27 is used for each FTP search problem representation in order to ensure that GA and random search techniques are given equivalent generation attempts in terms of fitness evaluations and FTP verification executions.

Two metrics are used to compare the results. The commonly used *state coverage* metric measures the number of states in M that have at least one FTP generated for every FTP size attempted. The *success rate* metric measures how many FTPs were generated compared to the total number of attempts it took to generate this results. This metric includes all the unsuccessful attempts to generate an FTP for the given search. It is important to note that both metrics include attempts to generate FTPs where a transition path exists in the abstracted FSM, but the transition guards lead to it having quality 0.

All the results exclude any attempts to generate unspecified transition paths i.e. paths that do not exist in the abstracted FSM (e.g. attempts to generate a TP of size 2 for a state reachable in at least 3 transitions are discarded).

Figure 4.46 represents a summary of the result averages. In general the results show that PB notation seems to perform the same or better than transition notation according to both metrics. In two instances the averaged FTP search results in Figure 4.46 show identical performance of the PB notation and transition notation methods, the transition notation attempt rate for the Inres protocol performs slightly better than the PB notation equivalent, but in the nine other instances the PB notation is the clear winner. This performance gap between the two notations is likely to increase as larger EFSMs are considered. This is due to the fact that the search space is geometrically related to the number of transitions in an EFSM. The search space for PB notation search is bound by the EFSM abstraction defined in the beginning of this chapter (based on number of input declarations and PBs).

For both metrics the two GA search algorithms clearly perform better than the random generation algorithm. This suggests that the fitness function helps guide a heuristic search for FTPs. As larger EFSMs and longer FTPs are considered, the heuristics seem to perform increasingly better than the random generation algorithm when given equal processing effort in terms of fitness evaluations and FTP verifications.

The state coverage metric is the easier one to satisfy. Not surprisingly the GAs found at least one FTP for every state in $M1$ and $M2$. This measure however discards all the unsuccessful attempts to generate a given FTP. Hence the success rate metric considers those unsuccessful attempts as well. The success rates generated were better than expected (75% for M_1 and 54% for M_2).

notation EFSM		PB Inres	transition Inres	PB Class 2	transition Class 2
State Coverage	GA1	100%	100%	98%	98%
	GA2	100%	78%	100%	94%
	Random	44%	17%	32%	27%
Success rate	GA1	75%	73%	54%	43%
	GA2	83%	45%	43%	40%
	Random	41%	19%	26%	27%

Figure 4.46: GA and Random search result averages for the Class 2 and Inres protocols in PB notation and transition notation.

The results for each of the EFSMs are discussed in more detail in the following two subsections. The results for both EFSMs show similar trends according to both metrics. The state coverage and success rate metrics seem to slightly differ between the two EFSMs. The state coverage and success rate results for M_1 and M_2 are similar for most of the GA generated results but the random algorithm generated results for the two EFSMs are slightly different. It is important to note that for Figures 4.47 to 4.53 the points represent result data and the connecting curved line are drawn just to illustrate the trend.

Class 2 Transport Protocol

The Class 2 Transport Protocol (M_2) is the larger of the two EFSMs we discuss. Figure 4.47 represents the state coverage for FTPs generated using PB notation.

GA1 has performs well and GA2 fails to find one FTP of size 4, while the random generation algorithm performance peaks when generating FTPs of size 2 and 3 and gradually declines as the FTP size increases.

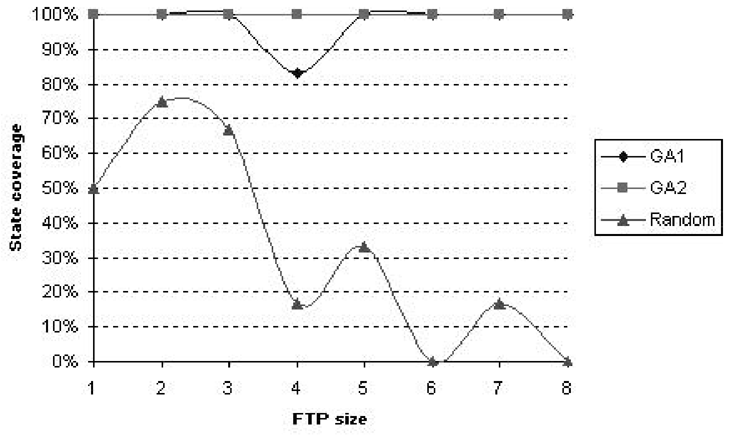


Figure 4.47: State coverage for PB notation FTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

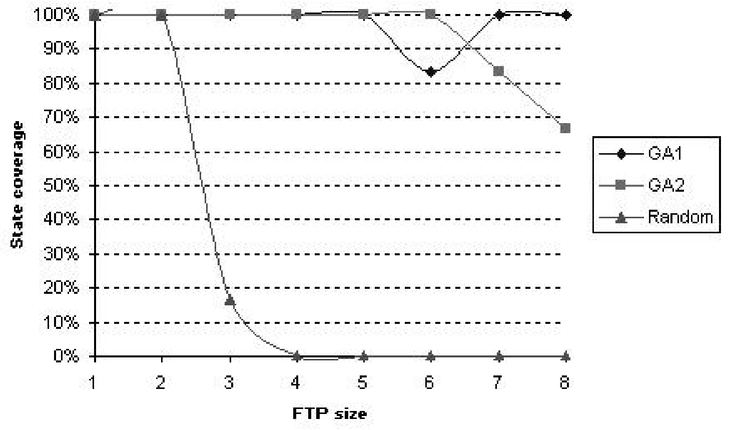


Figure 4.48: State coverage for transition notation FTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

Figure 4.48 represents the state coverage for FTPs generated using transition notation. GA1 performs well and only misses one FTP of size 6 while GA2 finds that

FTP but fails to find an FTP of size 7 and two of size 8. The random generation algorithm performance equals that of the GAs for FTPs of size 1 and 2 but as the size increases its performance sharply declines. There are no FTPs found by the random generation algorithm of length 4 to 8.

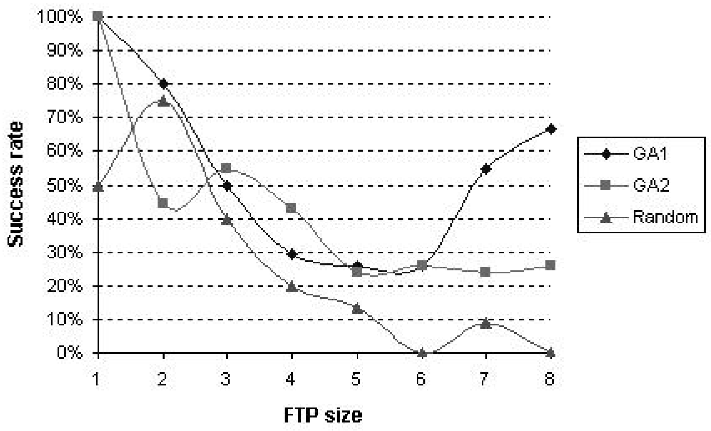


Figure 4.49: Success ratio for PB notation FTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

Figure 4.49 represents the success rate for FTPs generated using PB notation. The higher fluctuation rate here can be explained by the different degree of difficulty in generating FTPs of different sizes for some states. This relates to the future work on the fitness function discussed earlier in Section 4.5.2. GA1 shows the best performance with a peak performance of 100% and worst performance of 26%. GA2 performs mostly better than the random generation algorithm, except for FTPs of size 2. For FTPs of size 3 and 4 it even performs slightly better than GA1. The random generation algorithm performs slightly better than GA2 for FTPs of size 2, but the performance rapidly drops to towards 0% as the size of the FTPs increases.

Figure 4.50 represents the success rate for FTPs generated using transition notation. A similar effect between the GA1 and GA2 values can be observe here as with the PB notation attempt rate. However here the random generation algorithm starts

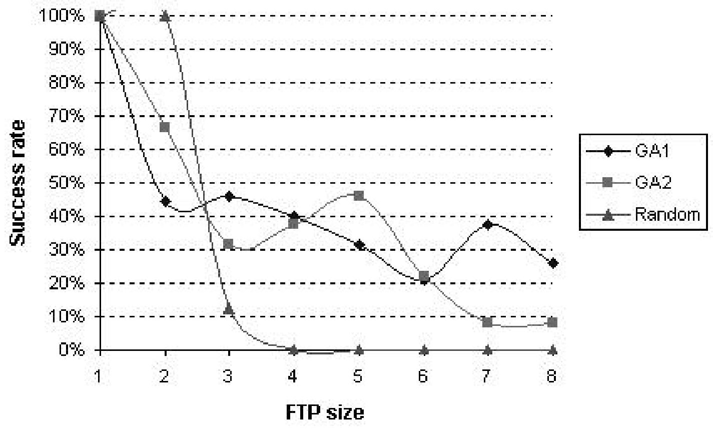


Figure 4.50: Success ratio for transition notation FTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

at a 100% for FTP sizes 1 and 2, but quickly drops to 0%. This excellent start for the random generation algorithm can be linked to the ease of generating very short FTPs since most valid transition paths of such length (initiating form the start state s_1) are likely to be feasible.

Inres Protocol

The Inres protocol EFSM (M_1) is smaller than M_2, but as the fitness evaluation results in Section 4.5.2 indicated not necessarily easier to generate FTPs for.

Figure 4.51 represents the state coverage for FTPs generated using PB notation. The two GAs have performed equally well, while the random generation algorithm performance equals the GAs FTPs of size 1 and 2 but declines as the FTP size increases. No FTPs with sizes 5, 7 or 8 were generated by the random generation algorithm.

Figure 4.52 represents the state coverage for FTPs generated using transition notation. GA1 performs well and GA2 fails to find some FTPs with sizes 4 and 7 and all FTPs of size 8. The random generation algorithm succeeds for only half of

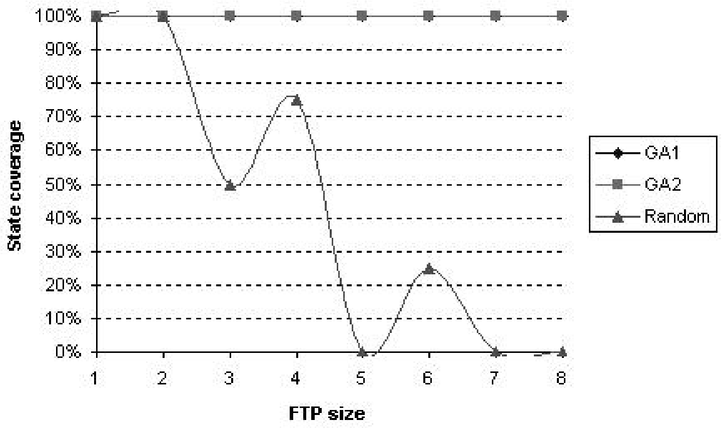


Figure 4.51: State coverage for PB notation FTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

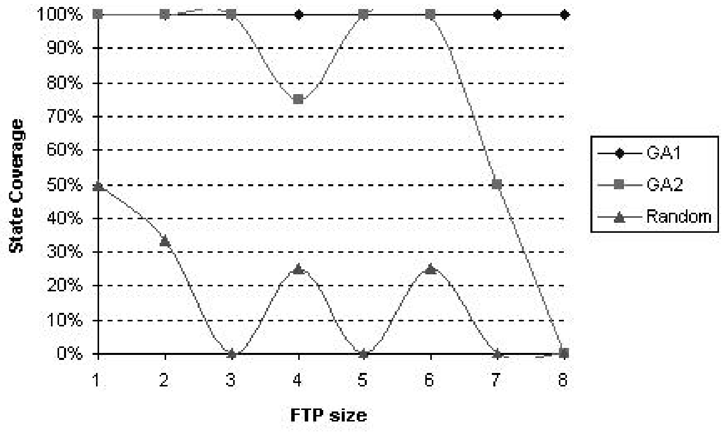


Figure 4.52: State coverage for transition notation FTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

the states for FTP size 1 and declines with some partial success for FTPs of size 4 and 6 but fails to generate FTPs with sizes 3, 5, 7 and 8.

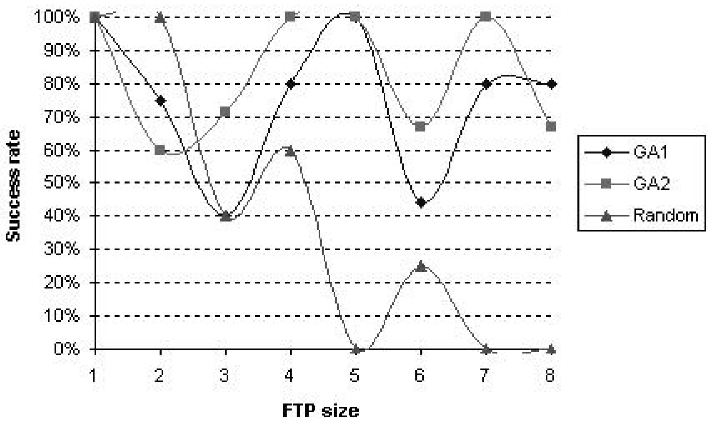


Figure 4.53: Success rate for PB notation FTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

Figure 4.53 represents the success rate for FTPs generated using PB notation. A similar effect between the GA1 and GA2 values can be observe here as with the attempt rate for M_2. When generating FTPs with size 2 GA1 takes one attempt more than it takes the random generation algorithm but GA1 generates FTPs with larger sizes with better efficiency than the random generation algorithm. When generating FTPs with size 2 GA2 took 3 attempts more than it took the random generation algorithm. The GA1 attempt rate is the same as the random generation algorithm for FTPs of size 3. GA2 seems to generate the best results overall. GA1 however generates FTPs of size 8 with better efficiency than GA1 and the random generation algorithm. The random generation algorithm starts at at a high 100% for FTPs of size 1 and 2 but steady decline to 0% for FTPs with sizes 5,7 and 8.

Figure 4.54 represents the success rate for FTPs generated using transition notation. GA1 seems to perform best although the attempt rate steadily declines from 100% for FTPs with size 1 and 2 until it picks up for FTPs with size 7 and 8. GA2

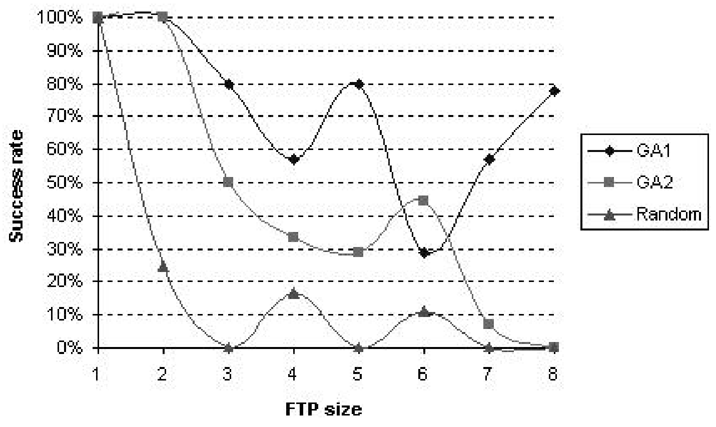


Figure 4.54: Success rate for transition notation FTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

also starts at 100% for FTPs with size 1 and 2 but declines to 0% for FTPs with size 8, but is slightly better than GA1 for FTPs of size 6. The random generation algorithm starts at 100% attempt rate for generation FTPs with sizes 1 but sharply drops to no FTPs generated for sizes 3, 5, 7 and 8.

4.6 Summary

The fitness evaluation results suggest that the principles used in the fitness function estimate the feasibility of a given transition path (TP) with reasonable accuracy. The 0.72 and 0.62 correlation factors suggest a good correlation between the fitness algorithm and estimated feasibility for the TPs for the two EFSMs tested.

The heuristic search results suggest that the novel EFSM abstraction using predicate branches (PBs) could help in the search for feasible transition paths (FTPs). Compared with the transition notation results, the PB notation searches generated better results for both EFSMs. It was also found that a GA that used the fitness function was more effective in FTP generation than a random algorithm.

Although the path could be generated using Dijkstra's shortest path algorithm it is important to remember that generating FTPs is the first stage of an attempt to generate feasible test transition paths for EFSMs and the input sequences to trigger them. Search for such test sequences may require information about an EFSM that could not be presented in appropriate form for Dijkstra's shortest path algorithm.

A particular focus for future work on FTPs is to improved fitness estimation for loops. Some more experimental work could further help increase the accuracy of the fitness algorithm.

Chapter 5

State Verification Transition Path generation for EFSMs

5.1 Introduction and motivation

As discussed in the motivation for Chapter 4 conformance testing for EFSMs is not simple. Typically the machines that arise are complex and exhaustive testing is infeasible [Lee 96]. Automating the generation of test sequences for EFSMs has thus been of a topic great interest [Lee 96, Li 94, Petr 04, Dual 04].

A system specified by an FSM or an EFSM is tested for conformance by applying a sequence of inputs and verifying that the corresponding sequence of outputs observed is that expected. Once a given transition is tested a state verification sequence can verify that the transition ended in the intended state. Unique input/output (UIO) sequences have been extensively used to solve this problem in FSMs. In Chapter 3 we explored the generation of UIOs. In Chapter 4 we discussed how feasible transition paths (FTPs) can be generated for EFSMs. Here we combine these two methods in an attempt to generate state verification transition paths (SVTPs) - transition paths

(TPs) that are easy to trigger and define state verification sequences for a given state of an EFSMs.

The objective of this work is to facilitate the generation of FTPs that can be used as state verification sequences. This can help in test data generation for EFSMs. In this chapter we focus on generating TPs that are likely to be feasible and whose observable behaviour can verify an end state. The overall approach to the problem is based on defining a fitness function that can estimate how likely it is that a transition path has both these properties. Then we can attempt to generate input sequences to trigger these transition paths.

Empirical data is used to evaluate the effectiveness of this fitness function. GAs are used to generate TPs and the results are evaluated by comparing the results to those produced using a random generation algorithm.

The chapter begins by outlining the problem. The approach used to search for SVTPs is described next. Then the fitness functions for the two different search problem representations for generating SVTPs are described as well as the algorithms used to verify potential solutions. The experimental strategy to collect the empirical evidence follows. Section 5.4 reports on the results of the experiment. Finally conclusions are drawn.

5.2 State verification transition paths (SVTPs)

In this section we define a state verification transition path (SVTP) and describe a fitness function that attempts to estimate how likely it is that a transition path is feasible (previously discussed in Section 4.3.1) and how likely it is that it represents a state verification sequence for a given state s in an EFSM M.

The ability to observe the internal behaviour of M is important at run time in order to verify that a TP is followed. For some EFSM based system implementations the values of the internal variables can be monitored at run time. We will refer to such systems as monitored systems. White box testing can be used to test such systems for conformance. For other EFSM based system implementations the values of the internal variables can not be monitored at run time. Such systems we refer to as non-monitored systems. Black box testing can be used to test such systems for conformance.

This work can be applied to both monitored and non-monitored system implementations when testing the implementation (M_I) of an EFSM based system M. For our experiment we simulate and monitor the execution of M (the specifications) in order to evaluate the effectiveness of our approach.

A transition t in an EFSM M is defined as (s_s, g_I, g_D, op, s_f) in Chapter 2. The observable behaviour of t is defined by g_I and *op*. The input declaration i and input parameter(s) P_i of the tuple g_I (i, P_i, g_{P^i}) are clearly observable since they are supplied by the user. The sequential operations *op* for t can consist of simple assignments and output statements. In FSMs the output generated by a transition is an output declaration from the known output alphabet or null (no observable output). However in EFSMs besides output declarations there are also output functions that take a number of parameters and generate an output based on these parameters.

There are two types of output functions in terms of observable behaviour. The output functions in communication protocols specified as EFSMs often represent an observable output parameter (output header) followed by the parameters that represent the transported data. In such cases the assignment statements in the *op* part of a transition will not generate any observable output and any two output functions will produce different headers and so different outputs.

Hence different output declarations are assumed to be distinguishable: this is similar to the 'output distinguishable' assumption used in the X-machines literature [Halc 98]. Two output functions $f_1(a_1)$ and $f_2(a_2)$ generate distinguishable output as they have different labels.

Definition 5.2.1. Two output function f_1 and f_2 are said to be output equivalent if there exists a_1 and a_2 such that $f_1(a_1) = f_2(a_2)$. Two output functions f_1 and f_2 are said to be *output-distinguishable* if for all a_1, a_2 we have that $f_1(a_1) \neq f_2(a_2)$.

Hence in order to establish output distinguishability for transitions with common output functions (e.g. $f_1(a_1)$ and $f_1(a_2)$) the values of the output parameters (a_1 and a_2) and the computation logic the function (f_1) must be known. When such information is unavailable $f_1(a_1)$ and $f_1(a_2)$ are assumed to be equivalent. In order to simplify the analysis this information is assumed unavailable.

Cases where two different output function $f_1 \neq f_2$ could be output equivalent $f_1(a_1) = f_2(a_2)$ for some values of a_1 and a_2 are not considered. Output-distinguishability cannot be guaranteed for such output functions. This topic remains one for future work.

Definition 5.2.2. A transition t initiating from state s is *observable* if there is no other transition t' initiating from s with the same input and equivalent output as t.

Definition 5.2.3. An EFSM M is observable if all of its transitions are observable.

Consider the Sending state (S_s) in the EFSM M_1 on Figure 4.1. There are two transitions initiating from that state that share the same input declaration AK, the same input parameter num and even the same input guard $num = number$, but have different domain guards. If M_1 is observable then these two transitions would generate different observable outputs. However they both do not generate any output. In order to generate test sequences for such EFSM examples in our work we have relaxed the observability constrain that is commonly used in similar work [Petr 04], by considering EFSMs that are not necessarily observable.

For example consider transitions t_4, t_7 and t_9 in M on Figure 4.1 that have output functions. t_7 and t_9 both start from the same state and have the same output function, therefore are assumed to be output equivalent. However t_4 is output distinguishable from the other transition leaving the same state - t_{13}.

Definition 5.2.4. Each output-distinguishable output function and output declaration can be represented by a unique symbol ω.

After defining output-distinguishability for output functions we can define a representation for each output function and each output declaration. In transitions where no output declaration or output function is present we can assume a null output.

t	out channel 1	out channel 2
t_a	ω_1	ω_2
t_b	ω_2	ω_1
t_c	ω_1	
t_d	ω_2	
t_e		ω_1
t_f		ω_2

Figure 5.1: Equivalent observable output assumption

Some EFSMs have a number of input and output channels (ports). Some transitions send output to more than one channel at a time. In some EFSM systems such output channels are not individually observable. For example consider the transitions in Figure 5.1. The listed transitions output to two channels and the observable output for t_c and t_e can be distinguished from that of t_d and t_f. By contrast the output of t_c and t_e in Figure 5.1 cannot be distinguished for such EFSM systems.

In this work we assume that each channel is separately observable, hence all of t_a, t_b, t_c, t_d, t_e and t_f can be distinguished from each other.

In FSMs verifying the end state of a transition is often considered a sufficient measure to identify potential transfer faults in the implementation. In EFSMs it is necessary to verify the configuration of the EFSM after a transition. An EFSM starts in its initial configuration and there is a configuration for every combination of state s and values of the set internal variables V.

Definition 5.2.5. A configuration for an EFSM M is a combination of state and values of the internal variables V of M. It is also a state in the unfolded machine of M represented as an FSM.

In their recent work [Petr 04] Petrenko et al. explored the idea of generating configuration confirming sequences. Such an input sequence for an EFSM in a given configuration would produce an output sequence different from a defined subset of other configurations for that machine. Hence the tail configuration can be verified after a transition. The potential problem of combinatorial complexity associated with exploring all the configuration of an EFSM has been addressed by deriving a confirming test sequence for a designated reference configuration and a given black list of typical faulty configuration, supplied by the tester.

The work in this chapter is different from the work on confirming configuration sequences in a number of ways. We use a slightly different EFSM model with weaker assumptions concerning input completeness and transition observability. In this work we attempt to solve the problem of finding a TP in an EFSM initiating from a state s that would produce an output sequence that uniquely identifies s as the start state for that TP. The problem of finding input to trigger this TP is also briefly addressed. The issue of verifying the values of the internal variables is not explicitly considered. However a certain level of internal variable checking occurs in cases when some guarded transitions are triggered and their feasibility is verified.

It is possible to extend the notion of a UIO that can verify a given state of an FSM, to a TP from state s (triggered by a parameterised input sequence) that generates a unique distinguishable output sequence that will distinguish s in an EFSMs M from all other states. Such a sequence is a state verification transition path (SVTP) for the state s of M.

Definition 5.2.6. A state verification transition path (SVTP) for the state s of M

is a transition path, triggered by a parameterised input sequence x, which produces a distinguishable output sequence y such that y cannot be produced in response to x from any state $s' \neq s$.

An SVTP can be used to verify whether the correct tail state has been reached after a transition.

5.2.1 Fitness function

It is useful to have an easy to execute fitness function that guides the search for TPs that are easy to execute and at the same time have observable behaviour (in terms of input and output) that can uniquely identify a state.

The problem of generating FTPs was discussed in Chapter 4. The method used there to estimate the feasibility of a TP is replicated here. Transitions are classified into four types *simple*, g_{P^i}, g_D and g_{P^i}-g_D. TPs with *simple* transitions are encouraged while the other types of transitions are penalised according to their guard conditions. Hence the search is directed towards generating a FTP.

An SVTP can represent TPs that involve *simple*, g_{P^i}, g_D and g_{P^i}-g_D transitions. If an SVTP consists of only *simple* transitions then it is guaranteed to be feasible. In that case there would be no need to consider input parameters and the values of the internal variables V of the EFSM. In the same way the feasibility of a transition path that does not consist of only simple transitions cannot be guaranteed.

Transitions can also be classified according to the characteristics of their observable input /output behaviour. Similar to the ranking of input /output pairs in Chapter 3 all input declarations, output declarations and output functions can be labelled in such a way that every unique observable input /output behaviour is listed only once. Hence output functions with equivalent observability, as defined earlier, are only paired with an input once. The frequency of each input and output label pairs is enumerated. Similar to the fitness function used in Chapter 3 to direct a search for UIOs, here we can penalise a TP according to frequency of the input /output label pairs involved. The summation of the frequency values of the input and output label pairs involved in the TP can be used to estimate the likelihood of that TP representing a state verification sequence.

All transitions are ranked according to the combined feasibility and uniqueness values. Both frequency values are scaled to give them equal weight in the ranking. A bias for one of the frequency values can be used in an attempt to fine tune the fitness algorithm. This is briefly examined in Section 5.4.2.

Shorter test sequences are usually used where possible to minimise test effort. Similar to the UIO generation problem with FSMs [Lee 96] assuming all transitions take approximately the same time, shorter SVTP are likely to minimise test efforts. However some transitions in EFSMs might take longer to execute than others (consider a *simple transition* and a g_{P^i}-g_D transitions with an output function). Even if the execution time difference is not significant some SVTP would require more effort to generate and apply than others. SVTP with conditional transitions are harder to generate than SVTP with only *simple* transitions and might involve difficult to satisfy conditions. Hence a slightly longer SVTP with only *simple* transitions could still be more useful than a short SVTP with conditional transitions, which are difficult to satisfy. Therefore the most efficient SVTP in terms of how easily they are generated and applied can be considered an SVTP with only *simple* transitions. Such an SVTP represents a UIO sequence for the abstracted FSM of the EFSM.

SVTPs with g_{P^i} transitions that do not depend on internal variables would be the next best choice, followed by those with g_{P^i} transitions that depend on internal variables and those with g_D transitions. SVTPs with g_{P^i}-g_D transitions would be the hardest to generate and apply because both input guard and domain guard predicates have to be satisfied for each g_{P^i}-g_D transitions in the TP.

Finding UIO sequences for FSMs is known to be NP-hard, without the added complexity of guards. UIO generation is a special case of the SVTP generation problem hence the SVTP generation problem must also be NP-hard.

In Chapter 3 the search for UIOs was represented as a GA search problem using an easy to execute fitness function. In Chapter 4 a scalable fitness function was defined that estimated the ease of execution of a TP in an EFSM. Combining the two fitness functions in a multiple objective optimisation problem and searching for TPs with both properties could benefit test data generation for EFSMs.

5.2.2 Representation

Two types of TP representation for EFSMs were defined in Chapter 4: *PB notation* and *transition notation*. An SVTP is a TP that has both the characteristics of an FTP and UIO for an EFSM. Hence the same two notations can be used here.

Similarly to the FTP generation problem in Chapter 4 the two different TP representations (*transition notation* and *PB notation*) require slightly different fitness functions and verification algorithms. Hence the two representations define two different SVTP search problem representations.

5.3 Algorithms

As with the FTP generation problem two SVTP search problem representations are presented. In order to compare them each one is used with all four SVTP generation techniques. The search techniques include two slightly different GAs, a random generation algorithm and BFS algorithm using the notation used in Chapter 4 for FTP search.

5.3.1 Preprocesses

A number of process must be completed before the fitness function and SVTP verification algorithm can be used for an EFSM. The five processes are shown on Figure 5.2. The table lists the complexity of each process and which of the two FTP representations that process facilitates. Similar to the work on FTPs in Chapter 4 the input enumeration and PB count processes are specific to the PB notation, while Input/Output ranking and feasibility ranking are used for both PB notation and transition notation.

	Process	complexity	Used for PB	transition not.
1	input/output ranking	$O(\lvert T\rvert.log\lvert T\rvert)$	✓	✓
2	feasibility ranking	$O(\lvert T\rvert.log\lvert T\rvert)$	✓	✓
3	Input enumeration	$O(\lvert T\rvert)$	✓	x
4	Distinguishable Output enumeration	$O(\lvert T\rvert.log\lvert T\rvert)$	✓	✓
5	PB count	$O(\lvert T\rvert)$	✓	x

Figure 5.2: Preprocesses used by the fitness and verification functions for FTP generation using the PB notation and transition notation. $|T| \geq n$ since we are looking at initially connected EFSMs

Input enumeration and PB count processes are the same as those in Chapter 4. Feasibility ranking is equivalent to the transition ranking process in Chapter 4.

The distinguishable output enumeration process is a simple enumeration of all the transitions of M, giving the same label to all output equivalent transitions. The results of this process is the ω column in Figure 5.3.

The only other process that is different from those used for FTP generation is the input/output ranking. This process is similar to the transition ranking process in Chapter 3 used for UIOs. This process ranks each input/output pair of the EFSM M according to how many times it reoccurs in the transition table of M. A pair that occurs only once gets the lowest rank, a pair that occurs twice is ranked next etc. Pairs that have the same number of occurrences in the transition table get the same rank.

Figure 5.3 shows the ranked transition table for the EFSM from Figure 4.1 (M_1). For example t_3 and t_{10} share the same input and ω_3 output pair and therefore are ranked lower than some other transitions. However t_{11},t_{12},t_{13} and t_{14} all share the same input and ω_3 output pair and are ranked even lower.

t	input	output	ω	feasibility rank	i/o rank
t_0	ICONreq	!CR	ω_1	0	0
t_1	CC	!ICONconf	ω_2	0	0
t_2	T_expired	!CR	ω_1	2	0
t_3	T_expired	!IDISind	ω_3	1	1
t_4	IDATreq	DT	ω_4	0	0
t_5	AK		ω_5	6	1
t_6	AK		ω_5	6	1
t_7	AK	DT	ω_6	5	0
t_8	AK	!IDISind	ω_3	4	0
t_9	T_expired	DT	ω_6	3	0
t_{10}	T_expired	!IDISind	ω_3	2	1
t_{11}	DR	!IDISind	ω_3	0	2
t_{12}	DR	!IDISind	ω_3	0	2
t_{13}	DR	!IDISind	ω_3	0	2
t_{14}	DR	!IDISind	ω_3	0	2

Figure 5.3: Input/output ranking and feasibility ranking for all transitions in M_1

5.3.2 Representation 1: Fitness function using PBs

The first SVTP search problem representation represents a transition using input characters and PBs. It has all the characteristics of the FTP search problem representation using PBs, but also considers the input/output ranking of the transitions involved.

An SVTP is considered invalid when M is in state s and there is no corresponding transition from s that can be triggered by the next input character in the generated input sequence. If a transition from the TP triggered by an AIS is invalid the EFSM stays in the same state and the next transition in the sequence is considered. Hence the whole AIS under consideration can be evaluated by the fitness function even when invalid transitions have been attempted.

This SVTP search problem representation has the following characteristics:

- The fitness function rewards highly ranked transition with less frequently used input and observable output (ω) pairs.
- It produces a numeric value potentially showing how close an input sequence is to defining a valid SVTP.
- The fitness value of the sequence incurs a penalty for each invalid transition.
- The GA is directed towards generating input sequences that contain mostly easy to execute transitions with unique input output behaviour and hence more likely to be feasible transitions that can verify a state.
- The fitness function represents the search for an SVTP sequence as a function minimisation problem so an AIS with a lower fitness value is considered to be more likely to form an SVTP since it is made up of more highly ranked transitions.

The fitness does not guarantee that a particular transition path can be triggered. It makes sure that it is constructed using consecutive transitions that are highly ranked. The verification process then checks if an IS can be generated to trigger such a TP.

The algorithm for the fitness function proposed is presented in Figure 5.4. The parameters involved for this function presented in Figure 5.5 are as follows: S_{SVTP} is the start state of the TP; min and max are the value range for randomly generated input parameters; $parList$ is the array of input parameters; l is the length of the input

```
$faults := 0$
$strengthValue := 0$
$S_k := S_{SVTP}$
if ($l = 0$) then return -1 //error code for empty SVTP
for($i := 1$ to $l$) //for all the transitions in the sequence
    $S_m := S_k$
    $S_k := \delta'(S_m, x_i, PB_i)$
    if ($S_k \neq \emptyset$)
        //There is such a transition
        $weight := bias2.R^{f}_{S_m,x_i,PB_i}.\frac{LCM(|R^f|,|R^{i/o}|)}{|R^f|} + bias1.R^{i/o}_{S_m,x_i,PB_i}.\frac{LCM(|R^f|,|R^{i/o}|)}{|R^{i/o}|}$
        $strengthValue := strengthValue + weight$
        //Equalising scale for both rankings and combining value
    endIf
    else
        $S_k := S_m$ //No such transition
        $faults := faults + 1$
    endElse
endFor
return $strengthValue + faults.penaltyValue1$
```

Figure 5.4: Representation 1: PB notation - SVTP fitness algorithm

S_{SVTP} - start state of TP
min, max value range for randomly generated input parameters
$parList$ - array of input parameters
C_V - the current values of all the variables in the internal variables set V of the EFSM
x - single input declaration
l - length of the input sequence
R^f - set of feasibility rankings
$|R^f|$ - set size
$R^f_{s,x,PB}$ - The feasibility rank for the transition initiating from state s with input declaration x and predicate branch PB
$R^{i/o}$ - set of input/output rankings
$R^{i/o}_{s,x,PB}$ - The input/output rank for the transition initiating from state s with input declaration x and predicate branch PB
$bias1$ - bias rate for the feasibility ranking
$bias2$ - bias rate for the input/output ranking
$penaltyValue1$ - penalty constant or function that penalises the fitness when an unspecified transition is triggered
$\delta'(s_i, x, PB) = s_j$ - EFSM abstracted state transfer function

Figure 5.5: Variables for the SVTP fitness algorithm using PB notation

sequence; R^f is the set of feasibility rankings; $R^f_{s,x,PB}$ represents the feasibility rank for the transition initiating from state s with input declaration x and predicate branch PB; $R^{i/o}$ is the set of input/output rankings; $R^{i/o}_{s,x,PB}$ represents the input/output rank for the transition initiating from state s with input declaration x and predicate branch PB; $bias1$ is the bias rate for the feasibility ranking; $bias2$ is the bias rate for the input/output ranking; $penaltyValue1$ is the penalty constant or function that penalises the fitness when an unspecified transition is triggered and $\delta'(s_i, x, PB) = s_j$ is the EFSM abstracted state transfer function. The parameters involved are also shown on Figure 5.5.

The SVTP verification algorithm for TPs generated in PB notation is analogous to the SVTP verification algorithm for transition notation described in Section 5.3.3.

5.3.3 Representation 2: Fitness function using transition notation

The alternative and simpler representation of the search problem for SVTP is transition notation. The algorithm for evaluating the fitness function used with this alternative TP representation is presented on Figure 5.6. The parameters involved are as follows: S_{SVTP} is the start state of the TP; T is the finite set of all transitions in M; T' is the transition sequence whose fitness is being determined; l is the length of the input sequence; R^f is the set of feasibility rankings; $R^f_{t_i}$ represents the feasibility rank transition t_i; $R^{i/o}$ is the set of input/output rankings; $R^{i/o}_{t_i}$ represents the input/output rank for transition t_i; $\delta(s_i, x, parList, C_V) = s_j$ is the EFSM state transfer function; $bias1$ is the bias rate for the feasibility ranking; $LCM(x, y)$ is the lowest common multiple of x and y; $bias2$ is the bias rate for the input/output ranking and $penaltyValue1$ is the penalty constant or function that penalises the fitness when an unspecified transition is triggered or the transitions are not consecutive (i.e. not following each other). The parameters involved are also shown on Figure 5.7.

Before this fitness algorithm can be used a number of processes must be completed. These include three of the processes used by the FTP fitness algorithm for PB notation - feasibility ranking, input /output ranking and distinguishable output enumeration. The general aim of the fitness function is to correctly sequence transition labels so that they form a TP that is likely to be an SVTP. Transitions are ranked according to the criteria described in Section 5.3.2. The function rewards higher ranked transitions and introduces a penalty when transitions are not sequential.

A transition from the transition sequence is considered invalid if it does not start from the state where the previous transition ended. If a transition is invalid the fitness value incurs a penalty and the next transition in the sequence is considered in turn by the algorithm. Hence the whole sequence of transitions under consideration can be

```
$faults := 0$
$strengthValue := 0$
$S_k := S_{SVTP}$
if ($l = 0$) then return -1 //error code for empty FTP
for($i := 1$ to $l$) //for all the transitions in $T'$
    $S_m := S_k$
    $S_k := \Pi_1(t_i)$ //Start state of transition $i$
    //if the transition initiates from the last reached state
    if($S_k = S_m$)
      $weight := bias2.R^f_{t_i}.\frac{LCM(|R^f|,|R^{i/o}|)}{|R^f|} + bias1.R^{i/o}_{t_i}.\frac{LCM(|R^f|,|R^{i/o}|)}{|R^{i/o}|}$
      $strengthValue := strengthValue + weight$
      //Equlising scale for both rankings and combining value
    endIf
    //Ignore this transition since it does not initiate from $S_m$
    else
        $S_k := S_m$
        $faults := faults + 1$
    endElse
endFor
return $strengthValue + faults.penaltyValue1$
```

Figure 5.6: Representation 2: Transition notation - SVTP fitness algorithm

S_{SVTP} - start state of SVTP
T - finite set of all transitions in M
T' - is the transition sequence whose fitness is being determined
l - length of the input sequence
$\delta(s_i, x, parList, C_V) = s_j$ - EFSM state transfer function
$\tau(s_i, x, parList, C_V) = t_j$ - EFSM transition identification function
$LCM(x, y)$ - lowest common multiple of x and y
R^f - set of feasibility rankings
$|R^f|$ - set size
$R^f_{t_i}$ - The feasibility rank for transition t_i
$R^{i/o}$ - set of input/output rankings
$R^{i/o}_{t_i}$ - The input/output rank for transition t_i
DoD^{max}_l - highest possible value of DoD for SVTPs of size l
$exec^{max}_l$ - highest possible value of $exec$ for SVTPs of size l
$bias1$ - bias rate for the feasibility ranking
$bias2$ - bias rate for the input/output ranking
$penaltyValue1$ - penalty constant or function that penalises the fitness when an unspecified transition is triggered or the transitions are not consecutive (i.e. not following each other)

Figure 5.7: Variables for the SVTP fitness and SVTP verification algorithm using transition notation

evaluated by the fitness function even when invalid transitions have been attempted.

Similar to the transition notation algorithm in Chapter 4 the fitness function represents search for an SVTP sequence as a function minimisation problem so a sequence of transitions with a lower fitness value is considered to be more likely to form an SVTP since it is made up of more highly ranked transitions.

SVTP verification is described in the next section.

5.3.4 SVTP verification

Similar to the FTP generation in Chapter 4, in order to evaluate the results of the experiment it is useful be able to estimate the quality of a TP. Such a quality measure can estimated how easy is it to trigger a TP or generate an IS that would trigger it.

When estimating a quality measure for FTPs in Chapter 4 we generated FTPs from the initial configuration for M_1 and M_2. Now we are interested in TPs initiating from all the states of M_1 and M_2. The ability to trigger some TPs is dependant on the configuration of M_I, which we cannot always monitor. A preamble is a short (often the shortest) path from the initial state to the transition to be tested. In order to attempt generation of SVTPs for all states in M_1 and M_2 we have chosen a set of the shortest simple (non conditional) TPs generated in Chapter 4 as our preambles to all TP generation attempts. This allows us to estimate the potential quality of TPs to represent SVTPs for only a limited number of configurations of M_1 and M_2, but the results should be enough to estimate the performance of this approach. An alternative way of overcoming this problem is to attempt to generate a pre-amble TP (or FTP) of arbitrary length for every SVTP generation attempt. Such alternative verification techniques and the measurement of their effectiveness remain future work.

We wish to estimate how easy is it to produce an IS for a TP in order to estimate the quality of TPs produced. This is done again by making 1000 attempts at randomly generating an IS for a TP from state s and calculating the success ratio. In addition to that the degree of difference (DoD) of the unique input /output behaviour, similar to the DoD defined in Chapter 3 and used to measure the quality of UIOs, of a TP for s is also calculated. This verification is stricter than the UIO verification used in Chapter 3. In EFSMs we have to compare the input /output behaviour of a TP to all other TPs from all states of the EFSM in order to ensure that it can be unique identify. This can lead to a worst case exponential problem due to there being multiple paths leaving each state. Instead we compare each transition of the TP individually with all other transitions in the EFSM and make sure that a TP has at least one transition with unique input /output behaviour in order to be an SVTP.

The DoD is used in order to measure the ability of a TP to uniquely verify a state. A positive value means there is at least one input /output pair that uniquely identifies the TP initiating from s. A TP with 0 DoD for s cannot unique identify s and hence cannot be an SVTP for s. Higher DoD value suggests higher fault tolerance for the SVTP (as discussed for UIO fault tolerance in Chapter 3). The more unique input /output pairs a TP contains, the lower the risk that a fault in the IUT M_I can mask all of them and render the SVTP unable to identify s for M_I.

The FTP verification algorithm is presented in Figure 5.8. The parameters involved are those used in Section 5.3.3 for the fitness algorithm with the following additional parameters: C_V represents the current values of all the variables in the internal variable set V of M; $\tau(s_i, x, parList, C_V) = s_j$ is the EFSM transition identification function; $exec_l^{max}$ is the highest possible value of $exec$ for SVTPs of size l and DoD_l^{max} is the highest possible value of DoD for SVTPs of size l. The parameters are summarised in Figure 5.7. The TP triggered in M is compared to the TP produced in SVTP generation. If the two TPs match the TP is considered to be feasible for that instant and that particular IS. For each TP the algorithm returns a value of 0 when no valid IS have been found to trigger that TP and a value of up to 1000 shows that some or all attempts succeeded. The maximum feasibility quality and maximum

DoD are scaled to match and these scaling factors are used to give equal weight to both quality measures when they are combined to produce the SVTP quality estimate. For example consider that the maximum quality factor for a TP with size l is 1000 and its maximum DoD can be 300. A TP with quality factor of 1000 and DoD of 150 would produce an SVTP quality estimate of 4500/6000 (1000 .3000/1000 + 150 .3000/300). The maximum SVTP quality estimate can be different for TPs with different lengths.

The FTP verification algorithm on Figure 5.8 checks if the correct TP is triggered (lines 1-20). Then the DoD for the TP is calculated for state s_{SVTP} (lines 21-43). When both values are positive a valid SVTP is found and the values are scaled to give equal weight to the SVTP quality estimate (line 44). If not explicitly mentioned the bias values are considered to be 1. Invalid SVTPs generate result 0 so that a valid SVTP can be easily identified by its value (lines 45-47). An SVTP value 0 would result from TP that produced 0 feasibility estimate or 0 DoD.

Section 5.4.2 describes the results of an empirical study on how SVTP verification results differ when the bias is altered.

5.4 Experiments

This section presents some empirical results on generating SVTPs for EFSMs M_1 and M_2. BFS algorithm is first used to generate a set of SVTPs for M_1 and M_2. The effectiveness of the fitness function to estimate the quality of an SVTP is evaluated using these results. Next the fitness function is used to guide heuristic search for SVTPs using the two different SVTPs search problem representations previously described. The strategy for generating these results is outlined in Section 5.4.1.

```
01 $exec := 0$ //feasibility estimate
02 $DoD := 0$ //degree of difference estimate
04 for ($repeats := 1$ to 1000)
05     $failed := false$
06     $S_j := S_{SVTP}$
07     for ($i := 1$ to $l$) //for all the transitions in $T'$
08         $parList := randomGeneration(min, max)$
09         $t_j := \tau(S_j, x_i, parList)$
10         $S_j := \delta(S_j, x_i, parList)$
11         $S_{temp} := S_j$
12         if ($t'_i \neq t_j$) //incorrect transition
13             $failed := true$
14         endIf
15         $S_j := S_{temp}$ //move to next transition
16     endFor
17     if ($failed = false$) //If no incorrect transitions
18         $exec := exec + 1$
19     endIf
20 endFor
21 $hasDifference := false$
22 for($k := 1$ to $l$)
23     $unique := true$
24     $in_k := \Pi_2(t_k)$ //Input declaration of transition $t_k$
25     $out_k := \Pi_4(t_k)$ //Distinguuishable output declaration of transition $t_k$
26     for ($m := 1$ to $|T|$)
27         if ($t_k \neq t_m$)
28             $in_m := \Pi_2(t_m)$ //Input declaration of transition $t_k$
29             $out_m := \Pi_4(t_m)$ //Distinguuishable output declaration of transition $t_m$
30             if ($in_k = in_m$ AND $out_k = out_m$)
31                 $unique := false$
32             else
33                 $DoD := DoD + 1$
34             endElse
35         endIf
36     endFor
37     if( $unique = true$)
38         $hasDifference := true$
39     endIf
40 endFor
41 if ($hasDifference = false$) //No unique input /output behaviour
42     $DoD := 0$
43 endIf
44 $success := bias1.exec.\frac{LCM(exec_l^{max}, DoD_l^{max})}{exec_l^{max}} + bias2.DoD.\frac{LCM(exec_l^{max}, DoD_l^{max})}{DoD_l^{max}}$
45 if($exec = 0$ OR $DoD = 0$) //Not a valid SVTP
46     $success := 0$
47 endIf
48 return $success$
```

Figure 5.8: SVTP Verification Algorithm

n - The number of states in the given EFSM
att - The number of attempts to generate an SVTP sequence with a specified length, start and end states to verify each state in the EFSM
min - The shortest SVTP to be generated
max - The longest SVTP to be generated
l - SVTP length being generated
i - the attempt number
s' - the initial state for each SVTP

Figure 5.9: Variables for the test strategy algorithm

5.4.1 Experiment strategy

Similar to the experiments in Chapter 4 the objective of the experiments is to evaluate the effectiveness of the fitness function defined and to attempt to generate SVTPs using heuristics.

When searching for a state verification TP one approach (used in Chapter 3 for UIOs) is to start from every state s_i in M and attempt to generate SVTPs of different lengths. Although this approach seems to work for FSMs (as used in Chapter 3) for EFSMs we have to consider all different configurations of an EFSM. Therefore we have used the same approach as for FTPs in Chapter 4. We attempt to generate an SVTP for every state in M for any given TP size. This approach is different to the one in Chapter 4 since we do not specify the end state of a TP.

The strategy for generating SVTPs using heuristic search is shown on Figure 5.10 and involves the following parameters: n is the number of states in the given EFSM; att is the number of attempts to generate an SVTP sequence with a specified length l and start state s' to be verified; min is the shortest SVTP to be attempted and max is the longest SVTP to be attempted. The parameters involved are also shown on Figure 5.9.

In contrast to the experimental work in Chapter 4, here we attempt to generate SVTPs from every state of M. Following the discussion in Section 5.3.4 we will use

```
for($s' := 1$ to $n$) //for all states in $M$
    for($l := min$ to $max$) //for all the SVTP lengths
        for($i := 1$ to $att$) //for all the repeated attempts
            $SVTP_{s',l}$ := attempt to generate SVTP with length $l$
            //Optional exit from additional attempts after an SVTP is found
            if($SVTP_{s',l}$ valid SVTP from $s'$ to verify state $s'$)
                $i := att$ //exit loop and move to next length
            endIf
        endFor
    endFor
endFor
```

Figure 5.10: Test strategy algorithm

the shortest FTPs from the initial configuration of M_1 and M_2 to all their states. The selected FTPs are listed in Figure 5.11. This is just one way of approaching this problem and required the least amount of additional work on the software tool used to generate the results. Using this method of exploring all states (but not all configurations) of M_1 and M_2 also enables us to use BFS algorithm and use the generated results to evaluate the effectiveness of the fitness function. In contrast to the work in Chapter 4 Dijkstra's shortest path algorithm could be used to generate only the shortest SVTP for a state s. If we select an alternative approach to explore all the states of M_1 and M_2, like generating a random pre-amble TP, Dijkstra's algorithm will no longer be useful for SVTP generation. The use of alternative state or configuration selection approaches remain future work.

By using M's initial state for all FTPs M would be in the same initial configuration (start state and values of the internal variables) with a simple reset. The problem of placing M in a given configuration before input sequence execution, other than its initial configuration, is not a focus of this work. A reset is not a necessary condition for the FTP generation but we assume a reliable reset for this test strategy in order to do multiple executions of M without considering the problems associated with placing M in a given configuration.

M	End State	TP
M_1	s_w	reset
M_1	s_w	t_0
M_1	s_c	t_0, t_1
M_1	s_s	t_0, t_1, t_4
M_2	s_1	reset
M_2	s_2	t_0
M_2	s_3	t_1
M_2	s_4	t_0, t_2
M_2	s_5	t_0, t_3
M_2	s_6	t_0, t_2, t_{17}

Figure 5.11: M_1 and M_2 pre-amble FTPs

The algorithm attempts to generate an SVTP from state s' in M for length ranging from *min* to *max*. No more than *att* number of attempts are made for every SVTP length. In order to generate comparable results, given an EFSM the *att*, *min* and *max* attributes are kept the same for the different heuristic SVTP generation techniques - GAs and Random. For each different SVTP generation technique the same test strategy is used but the appropriate FTP search problem representation used (GA with PB notation, GA with transition notation and random generation). SVTP generation using the BFS algorithm does not use the test strategy in Figure 5.10 as the algorithm finds all paths of a specified length between a given state s and all states in M.

5.4.2 Fitness evaluation results

This section describes results of a limited empirical study on the effectiveness of the proposed fitness algorithm. This is done by measuring the effectiveness of the proposed fitness algorithm in estimating the quality of a TP in an EFSM of being an SVTP for a given state s. Similar to Chapter 4 the study is limited to two

EFSMs (Inres protocol M_1 and a Class 2 transport protocol M_2). TPs of length 1 to l, initiating from each state in M_1 and M_2 were generated using BFS algorithm. The TP generation was irrespective of notation since transition and predicate branch notations represent the same TPs. For ease of presentation the TP examples are shown in transition notation.

The fitness function attempts to estimate how easily a TP can be executed and how effective it is at verifying state s (hence how easily can the input to trigger this TP be generated). The evaluation of the quality factor of an SVTP is a combination of the feasibility estimate (1000 attempts to execute that TP with random input parameters) and the DoD estimate (how unique is the input /output behaviour produced). A negative statistical correlation is expected between the fitness values and the TP quality factor values. Since the DoD element of the SVTP estimate is different for every SVTP size we have summarised the results by SVTP size.

Class 2 Transport Protocol

The Class 2 transport protocol M_2 is presented in Figure 4.28 and the corresponding transition table is shown in Table 5.12 (the guards were shown in Chapter 4 in Table 4.30). M_2 has more states, transitions and is more complex than M_1. M_2 is a major module (based on the AP-module [Boch 90]) of a simplified version of a class 2 transport protocol [Rama 03]. M_2 represents only the core transitions of that EFSM, as used in [Rama 03].

1116 TPs were generated and plotted in Figures 5.13, 5.14, 5.15 and 5.16 that represent all possible TPs with length of up to 4 transitions. 735 of these TPs have positive SVTP quality factor (i.e. every TP was successfully triggered at least once in 1000 attempts and it has DoD > 0). All the 381 TPs with 0 SVTP estimate are drawn in Figures 5.13, 5.14, 5.15 and 5.16 using lightly shaded squares. As explained

t	s_{start}	s_{end}	i	Output	feasibility rank	i/o rank
t_0	s_1	s_2	U?TCONreq	N!TrCR	0	0
t_1	s_1	s_3	N?TrCR	U!TCONind	0	0
t_2	s_2	s_4	N?TrCC	U!TCONconf	3	0
t_3	s_2	s_5	N?TrCC	U!TDISind N!TrDR	4	0
t_4	s_2	s_1	N?TrDR	U!TDISind N!terminated	0	0
t_5	s_3	s_4	U?TCONresp	N!TrCC	1	0
t_6	s_3	s_6	U?TDISreq	N!TrDR	0	1
t_7	s_4	s_4	U?TDATAreq	N!TrDT	2	0
t_8	s_4	s_4	N?TrDT	U!DATAind N!TrAK	3	0
t_9	s_4	s_4	N?TrDT	U!error N!error	3	0
t_{10}	s_4	s_4	U?U_READY	N!TrAK	0	0
t_{11}	s_4	s_4	N?TrAK		6	1
t_{12}	s_4	s_4	N?TrAK	U!error N!error	4	1
t_{13}	s_4	s_4	N?TrAK		7	1
t_{14}	s_4	s_4	N?TrAK	U!error N!error	5	1
t_{15}	s_4	s_4	N?Ready	U!Ready	2	0
t_{16}	s_4	s_5	U?TDISreq	N!TrDR	0	1
t_{17}	s_4	s_6	N?TrDR	U!TDISind N!TrDC	0	0
t_{18}	s_6	s_0	N?terminated	U!TDISconf	0	0
t_{19}	s_5	s_0	N?TrDC	N!terminated U!TDISconf	0	0
t_{20}	s_5	s_0	N?TrDR	N!terminated	0	0

Figure 5.12: Transition table for M_2 excluding transition guards

in Section 5.3.4 TPs with 0 SVTP estimate cannot be guaranteed to be feasible (according to our feasibility estimation) or do not possess unique input /output qualities. Although the fitness aims to estimate feasibility and unique input /output behaviour, a high proportion of TPs failed this verification for this EFSM. However all such TPs with 0 quality factor have comparatively high fitness values.

This high proportion of TPs that cannot be guaranteed to be feasible was already observed in the results in Section 4.5.2 and will not be further addressed in this Chapter.

The fitness function seems to correctly estimate the SVTP quality of most of the 1116 TPs. There is an averaged (of the TPs for lengths 1 to 4) negative correlation factor of 0.89 between the fitness function and the quality factor illustrated. If we only consider the 735 SVTPs the averaged correlation factor is 0.56.

The FTP results (including unsuccessful TPs) are classified in four sets according to the size of the SVTPs. Figure 5.13 shows the results for all SVTPs of size 1. Since each SVTP consists of only a single transition, the data is quite concentrated. A correlation factor of 0.91 was observed and a factor of 0.56 when only SVTPs are considered.

Figure 5.14 shows the results for all SVTPs of size 2. The SVTPs found here seem to be scattered in and above mid-range values for the SVTP estimate. A correlation factor of 0.88 was observed and a factor of 0.55 when only SVTPs are considered.

Figure 5.15 shows the results for all SVTPs of size 3. The SVTPs found here also seem to be scattered in and above mid-range values for the SVTP estimate but seem to concentrate slightly more around the mid-point of 3000. A correlation factor of 0.88 was observed and a factor of 0.55 when only SVTPs are considered.

Figure 5.16 shows the results for all SVTPs of size 4. The SVTPs found here seem to be scattered in and above mid-range values for the SVTP estimate again but here they seem to concentrate even more around the mid-point of 2000 (for this size).

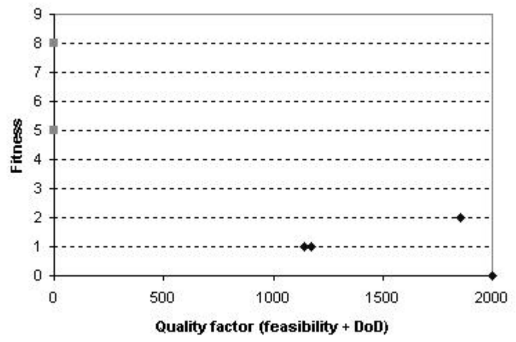


Figure 5.13: All SVTPs generated using BFS algorithm for M_2 of size 1 (correlation factor 0.91, when excluding the 0 quality factor TPs the correlation factor is 0.57). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

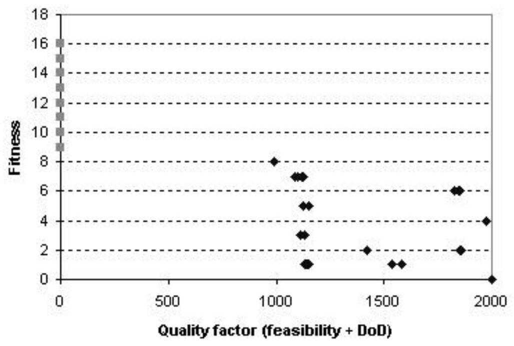


Figure 5.14: All SVTPs generated using BFS algorithm for M_2 of size 2 (correlation factor 0.88, when excluding the 0 quality factor TPs the correlation factor is 0.55). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

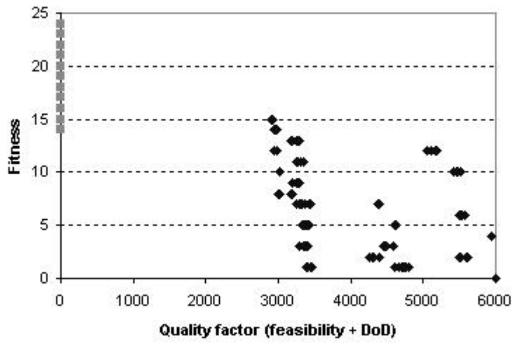


Figure 5.15: All SVTPs generated using BFS algorithm for M_2 of size 3 (correlation factor 0.88, when excluding the 0 quality factor TPs the correlation factor is 0.55). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

A correlation factor of 0.9 was observed and a factor of 0.58 when only SVTPs are considered.

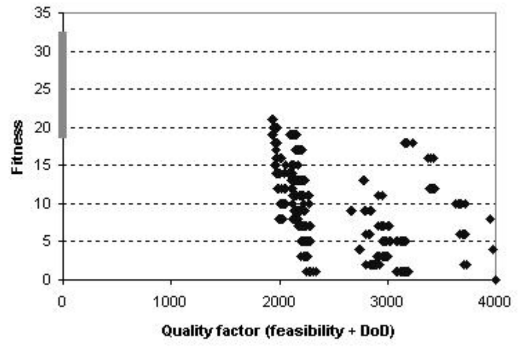


Figure 5.16: All SVTPs generated using BFS algorithm for M_2 of size 4 (correlation factor 0.9, when excluding the 0 quality factor TPs the correlation factor is 0.58). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

There is at least reasonable correlation between the SVTP estimate and the fitness values. When all TPs generated were considered the correlation is even stronger. This suggests that the fitness function can reasonably estimate the SVTP quality for a TP without actually executing it.

Inres Protocol

The EFSM M_1 in Figure 4.1 representing the Inres protocol that is simpler than M_2. The transition table for M_1 is presented in Figure 4.39.

BFS algorithms was used to generate 314 TPs for M_1 where only 12 were not SVTPs. Figures 5.17, 5.18, 5.19 and 5.20 illustrate those TPs that represent all possible TPs with length of up to 4 transitions with positive quality factor (i.e. every TP was successfully triggered at least once in 1000 attempts and it has DoD > 0). Again TPs with quality factor of 0 that cannot be guaranteed to be feasible (according to our feasibility estimation) or do not possess unique input /output qualities are drawn with lighter shaded squares. Here however there are considerably fewer TPs with 0 quality factor for M_1. This could indicate that the SVTPs, as it was for FTPs, are easier to generate for M_1.

The fitness function seems to correctly estimate the SVTP quality of most of the 314 TPs. There is an averaged (of the TPs for lengths 1 to 4) negative correlation factor of 0.59 between the fitness function and the quality factor illustrated. If we only consider the 314 SVTPs the averaged correlation factor is 0.66.

The SVTP results (including unsuccessful TPs) are classified in four sets according to the size of the SVTPs. Figure 5.17 shows the results for all SVTPs of size 1. Since each SVTP consists of only a single transition, the data is again quite concentrated. A correlation factor of 0.62 was observed and a factor of 0.95 when only SVTPs are considered. This sharp contrast in the correlation factors is caused by the 3 TPs with high fitness and 0 SVTP value out of 9 TPs of size 1 generated in total.

Figure 5.18 shows the results for all SVTPs of size 2. The SVTPs found here seem to be concentrated around the high end values for the SVTP estimate. A correlation factor of 0.59 was observed and a factor of 0.61 when only SVTPs are considered.

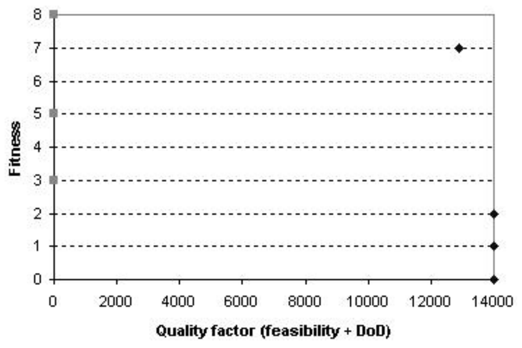


Figure 5.17: All SVTPs generated using BFS algorithm for M_1 of size 1 (correlation factor 0.62, when excluding the 0 quality factor TPs the correlation factor is 0.95). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

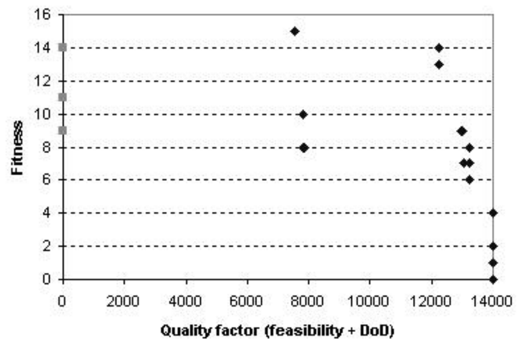


Figure 5.18: All SVTPs generated using BFS algorithm for M_1 of size 2 (correlation factor 0.59, when excluding the 0 quality factor TPs the correlation factor is 0.61). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

Figure 5.19 shows the results for all SVTPs of size 3. The SVTPs found here also seem to be concentrated in the mid-range and high end values for the SVTP estimate. A correlation factor of 0.58 was observed and a factor of 0.58 when only SVTPs are considered.

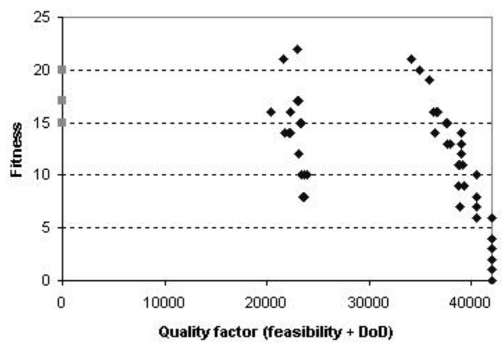


Figure 5.19: All SVTPs generated using BFS algorithm for M_1 of size 3 (correlation factor 0.58, when excluding the 0 quality factor TPs the correlation factor is 0.58). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

Figure 5.20 shows the results for all SVTPs of size 4. The SVTPs found here seem to be scattered in and above mid-range values for the SVTP estimate again but here they seem to concentrate around the mid-range SVTP estimate values. A correlation factor of 0.55 was observed and a factor of 0.51 when only SVTPs are considered.

There seems to be a reasonable correlation between the SVTP estimate and the fitness values here as well. When all TPs generated were considered the correlation is very similar due to the low number of 0 quality factor TPs. This suggests that the fitness function can reasonably estimate the SVTP quality for a TP without actually executing it for M_1 too.

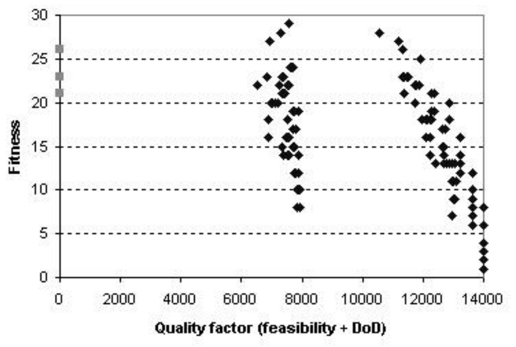


Figure 5.20: All SVTPs generated using BFS algorithm for M_1 of size 4 (correlation factor 0.55, when excluding the 0 quality factor TPs the correlation factor is 0.51). Light shaded squares represent the 0 SVTP estimate valued TPs. The dark diamonds represent SVTPs.

Introducing bias in fitness and verification algorithms

The fitness evaluation results presented so far have not used any bias (i.e. $bias1 = 1$ and $bias2 = 1$ in Figures 5.4, 5.6 and 5.8). In order to examine the effects of biasing the fitness algorithm and the verification algorithm additional results with bias towards the feasibility estimate or the uniqueness estimate were generated and presented on Figure 5.21.

The results in Figure 5.21 represent the fitness and quality estimate correlation for two bias scenarios. The first scenario presents a summary of the results for M_1 and M_2 where $bias1 := 2$, which doubles the weight of the feasibility value of a TP in the fitness ranking and feasibility estimate for SVTPs. The second scenario looks at the same correlation values but when $bias2 := 2$, which doubles the weight of the uniqueness value of a TP in the fitness ranking and feasibility estimate for SVTPs.

Similar to the results presented so far, we have considered the correlation for all TPs generated (including those with 0 SVTP quality) and the correlation for only the SVTPs.

When the results for M_1 are biased towards the feasibility estimate of a TP the

bias	SVTP size	Inres - M_1 incl. 0 (TPs)	Inres excl. 0	Class 2 - M_2 incl. 0 (TPs)	Class 2 excl. 0
none	1	0.62	0.95	0.91	0.57
	2	0.59	0.61	0.88	0.55
	3	0.58	0.58	0.88	0.55
feasibility - $bias1 := 2$	1	0.89	0.87	0.87	0.57
	2	0.75	0.58	0.85	0.19
	3	0.59	0.48	0.84	0.9
uniqueness - $bias2 := 2$	1	0.5	0.85	0.92	0.57
	2	0.34	0.6	0.9	0.65
	3	0.39	0.6	0.89	0.9

Figure 5.21: Summary of fitness and quality values correlation for SVTPs of sizes 1 to 3 for M_1 and M_2 using bias. All correlation values are negative and rounded to two decimal places

correlation for all the TPs generated is better than the original non biased results. However the correlation for all the SVTPs is worse when the bias is used.

When the results for M_2 are biased towards feasibility estimate of a TP the correlation for the TPs and SVTPs is worst than the original non biased results. The only two exceptions are the correlation for the SVTPs of size 1, when the correlation is the same and SVTPs of size 3 when the biased results have shown a considerably better correlation.

When the results for M_1 are biased towards the uniqueness measure of a TP, the results are slightly different. The biased results show worse correlation except for the SVTP generated of size 3, when the biased correlation is better.

When considering the results for M_2 where the uniqueness measure bias is used, all the correlations are better than those of the original non biased results.

Although it appears that biasing the fitness and quality measure towards the uniqueness measure or the feasibility measure of a TP can produce better correlation in some cases, it is important to note that as longer TPs are considered the ease of execution is likely to become more difficult to accomplish. This is also shown in the results where the feasibility biased correlations come close to or surpass those of the original non biased results as the size of the TPs considered grows. Therefore using a common bias for TPs of different lengths might not be the optimal solution. Instead a variable bias could be used that is related to the size of the TP considered.

Overall the non biased results performed well for both EFSMs. Hence unbiased results were used in an attempt to balance the influence of both factors. The effects of variable bias remains a topic for future work.

5.4.3 Generating SVTPs using Genetic Algorithm

Some positive results were generated while evaluating the fitness function hence we can use the test strategy in Figure 5.10 to attempt to generate SVTPs. Both SVTP search problem representations are used to generate SVTPs using the two GA and a random generation search techniques outlined earlier and also used in Chapter 4. The results are compared and conclusions drawn. The test strategy in Figure 5.10 is used for each SVTP search problem representation in order to ensure that GA and random search techniques are given equivalent generation attempts in terms of fitness evaluations and SVTP verification executions.

The same two metrics used in Chapter 4, *state coverage* (number of cases where at least one SVTP was generated for every SVTP size attempted from each state in M) and *success rate* (number of SVTPs that were generated compared to the total number of attempts it took to generate the results), are used to compare the results.

All the results exclude any attempts to generate unspecified transition paths i.e. paths that do not exist in the abstracted FSM (e.g. attempts to generate a TP of size 2 for a state reachable in at least 3 transitions are discarded).

Figure 5.22 represents a summary of the result averages. In general the results show that PB notation seems to perform better than transition notation according to both metrics. In only one instance in Figure 5.22 does the transition notation show slightly better performance than the PB notation, while the PB notation seems to perform up to 25% better. Similar to the results in Chapter 4 we expect this performance gap between the two notations to increase as larger EFSMs are considered. The reason for this is that the search space is geometrically related to the number of transitions in an EFSM. The search space for PB notation search is bound by the EFSM abstraction defined in the beginning of this chapter (based on number of input declarations and PBs).

For both metrics the two GA search algorithms clearly perform better than the random generation algorithm. Again like the results in Chapter 4 for FTP search this suggests that the fitness function here helps guide a heuristic search for SVTPs. As larger EFSMs and longer SVTPs are considered, the heuristics seem to perform increasingly better than the random generation algorithm when given equal processing effort in terms of fitness evaluations and SVTP verifications.

The state coverage metric is the easier one to satisfy. Not surprisingly the GAs found at least one SVTP for every state in M_1 and M_2. This measure however discards all the unsuccessful attempts to generate a given SVTP. Hence the success rate metric considers those unsuccessful attempts as well. The success rates generated were slightly lower than the ones in Chapter 4, but are still higher than expected considering the search problem is more challenging than FTP generation (48% for M_1 and 56% for M_2).

The results for both EFSMs show similar trends according to both metrics. The state coverage and success rate metrics seem to slightly differ between the two EFSMs. The state coverage and success rate results for M_1 and M_2 are similar for most of the GA generated results but the random algorithm generated results for the two EFSMs are slightly different. It is important to note that for Figures 5.23 to 5.29 the points represent result data and the connecting curved line are drawn just to illustrate the trend.

notation EFSM		PB Inres	transition Inres	PB Class 2	transition Class 2
State Coverage	GA1	97%	91%	100%	98%
	GA2	100%	75%	96%	77%
	Random	56%	28%	35%	29%
Success rate	GA1	54%	56%	48%	46%
	GA2	65%	40%	47%	31%
	Random	51%	26%	28%	23%

Figure 5.22: GA and Random search result averages for the Class 2 and Inres protocols in PB notation and transition notation.

The results for each of the EFSMs is discussed in more detail in the following two subsections.

Class 2 Transport Protocol

The Class 2 Transport Protocol (M_2) is the larger of the two EFSMs. Figure 5.23 represents the state coverage for SVTPs generated using PB notation. GA1 performs well and GA2 fails to find one SVTP of size 4 and one of size 8, while the random generation algorithm performance peaks when generating FTPs of size 4 and declines as the FTP size increases.

Figure 5.24 represents the state coverage for SVTPs generated using transition notation. GA1 performs well and only misses one FTP of size 6 while GA2 fails to find several SVTPs of sizes 2,5,6,7 and 8 but still performs better than the random generation algorithm. The random generation algorithm performance equals that of the GAs for SVTPs of size 1 but as the size increases the performance sharply declines. There are no SVTPs found by the random generation algorithm of length 6 to 8.

Figure 5.25 represents the success rate for SVTPs generated using PB notation. The higher fluctuation rate here can be explained by the different degree of difficulty in generating SVTPs of different sizes for some states. This relates to the future work

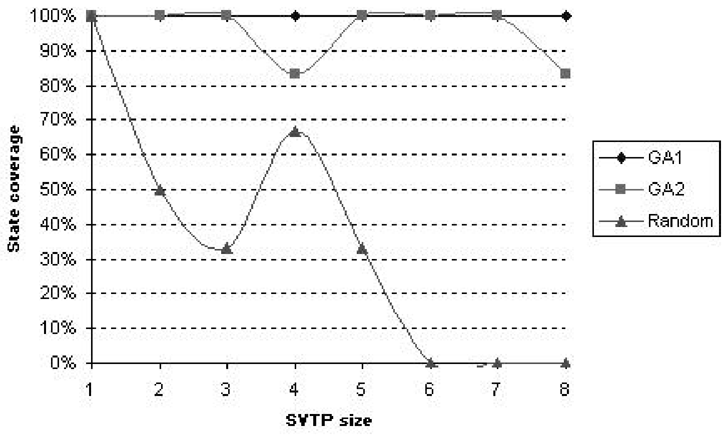


Figure 5.23: State coverage for PB notation SVTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

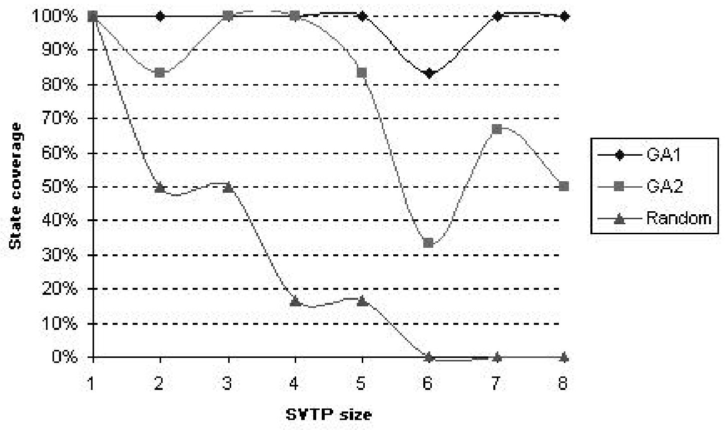


Figure 5.24: State coverage for transition notation SVTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

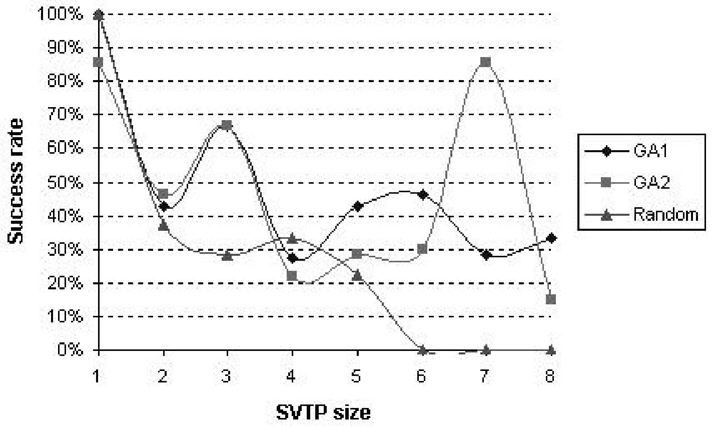


Figure 5.25: Success ratio for PB notation SVTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

on the SVTP fitness function discussed earlier in Section 4.5.2. GA2 shows the best performance with an average performance of 65%. GA1 performs mostly better than the random generation algorithm, except for FTPs of size 4 (average performance of 54%). For SVTPs of size 4 the random algorithm performs slightly better than GA1 and GA2.

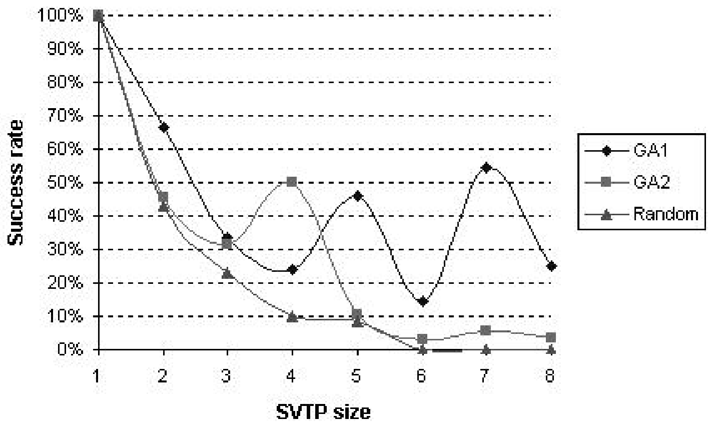


Figure 5.26: Success ratio for transition notation SVTPs generated using GA and Random generation algorithms for M_2 with 1-8 transitions

Figure 5.26 represents the success rate for SVTPs generated using transition notation. A similar effect between the GA1 and GA2 values can be observe here as with the PB notation attempt rate. However here the random generation algorithm starts at 100% for SVTP sizes 1, but quickly drops to 0%. This excellent start for the random generation algorithm can be linked to the ease of generating very short FTPs since most valid transition paths of such length (initiating form the start state s_1) are likely to be feasible. The GAs both averaged nearly twice the performance of the random generation algorithm.

Inres Protocol

The Inres protocol EFSM (M_1) is smaller than M_2, but as the fitness evaluation results in Section 5.4.3 indicated not necessarily easier to generate SVTPs for.

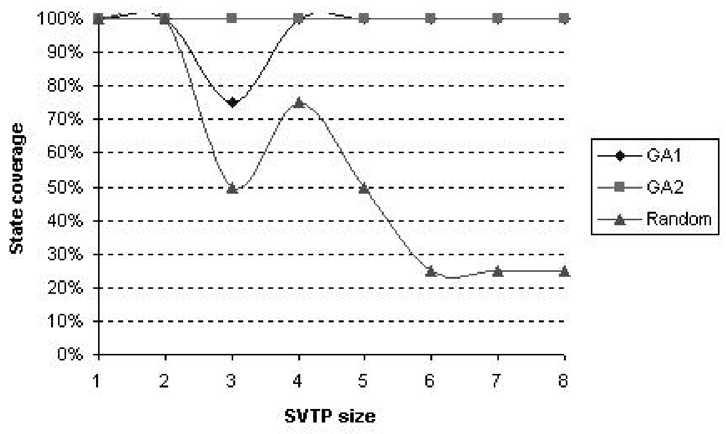


Figure 5.27: State coverage for PB notation SVTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

Figure 5.27 represents the state coverage for SVTPs generated using PB notation. GA1 performed well and GA2 failed to find one SVTP of size 3, while the random generation algorithm performance equals the GAs SVTPs of size 1 but declines as the SVTP size increases.

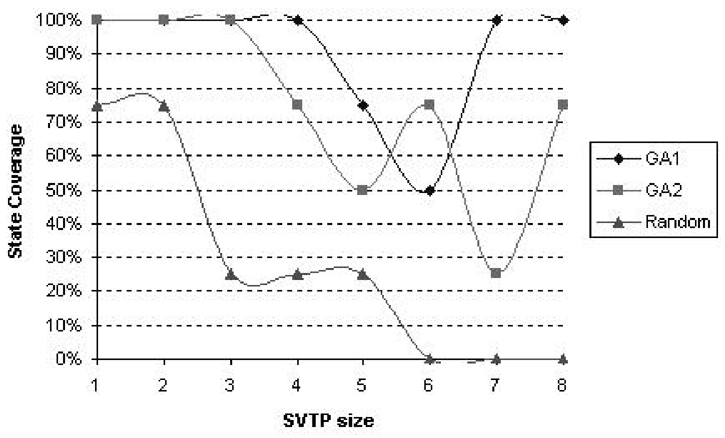


Figure 5.28: State coverage for transition notation SVTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

Figure 5.28 represents the state coverage for SVTPs generated using transition notation. GA1 performs well but fails to generate a small number of SVTPs of sizes 5 and 6. GA2 performs well for SVTPs of sizes 1 to 3 but fails to find some of the SVTPs of other sizes. The random generation algorithm generates SVTPs for 75% of the states of sizes 1 and 2, but as size increases it gradually fails to generate any SVTP of sizes 6, 7 and 8.

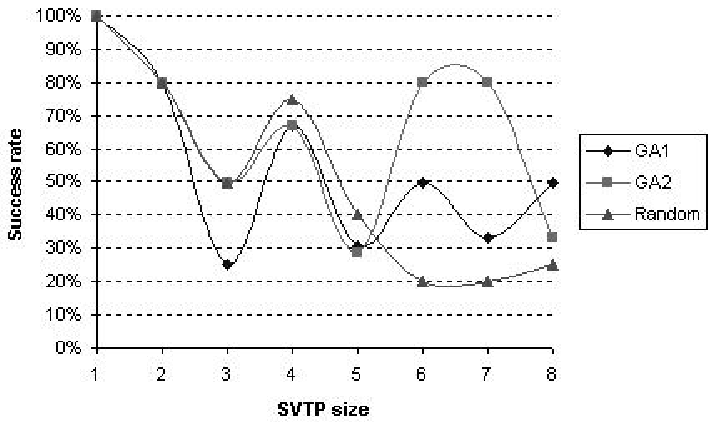


Figure 5.29: Success rate for PB notation SVTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

Figure 5.29 represents the success rate for SVTPs generated using PB notation. The GAs and the random generation algorithm have very similar success rate for SVTPs of sizes 1 to 5, where the random generation algorithm seems to have a slight lead. However for SVTPs of sizes 6, 7 and 8 both GAs perform considerably better than the random generation algorithm, especially GA2.

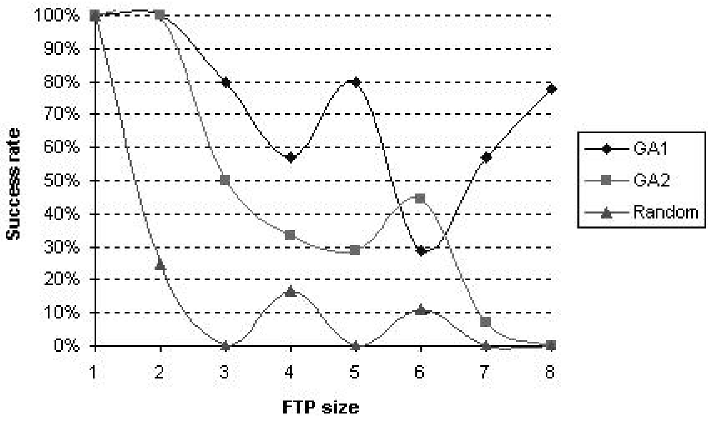


Figure 5.30: Success rate for transition notation SVTPs generated using GA and Random generation algorithms for M_1 with 1-8 transitions

Figure 5.30 represents the success rate for SVTPs generated using transition notation. The three methods follow the same shape of the success graph for SVTPs of sizes 1 to 5, but this time the GAs perform marginally or considerably better for those sizes. For SVTPs of size 6,7 and 8 the GAs clearly perform better while the random generation algorithm fails to find any SVTP.

5.5 Summary

The fitness evaluation results suggest that the principles used in the fitness function estimate the combination of feasibility and unique observable behaviour of a given transition path (TP) with reasonable accuracy. The 0.89 and 0.59 correlation factors suggest a good correlation between the fitness algorithm and estimated SVTP quality for the TPs for the two EFSMs tested.

The heuristic search results suggest that combining the search for feasible transition paths (FTPs), examined in Chapter 4, and the search for unique input /output sequences (UIOs), in Chapter 3, could help in the search for state verification transition paths (SVTPs). Compared with the transition notation results, the PB notation searches generated better results for both EFSMs. It was found that a GA that used the fitness function was more effective in SVTP generation than a random algorithm.

Although the shortest path could be generated using Dijkstra's shortest path algorithm it is important to remember that we used a simple EFSM configuration technique that allowed us to use our existing tool and estimate the correlation of the overall approach. An important advantage of using heuristic search to generate SVTPs, guided by a fitness function, is that the flexibility exists to add additional features to the fitness function. Future work on generating SVTPs and similar TPs seeks to use such features that could not be presented in appropriate form for Dijkstra's shortest path algorithm.

A particular focus for future work on SVTPs is, as it was for FTPs, to improved fitness estimation for loops. Some more experimental work could further help increase the accuracy of the fitness algorithm.

Chapter 6

Conclusions and future work

This thesis outlines the significance of automating the generation of test cases for FSMs. It evaluates the hypothesis that some test case adequacy criteria in FSM based systems can be estimated fairly accurately with efficient algorithms and GAs can use these estimations to generate such test cases that are otherwise NP-hard problems.

The thesis addresses three problems that are highly relevant to test data generation for FSMs and EFSMs, these being the efficient generation of UIOs for FSMs, estimating the feasibility of transition paths in EFSMs and the generation of state verification transition paths for EFSMs.

State verification is an important part of conformance testing of FSMs. UIO sequences are commonly used for state verification because of their advantages over other methods, but the problem of generating UIOs is NP-hard. GAs are known to perform well for some NP-hard problems.

Chapter 3 defines a fitness function of $O(l)$ complexity for an input sequence of size l that is used to guide a GA search to generate UIOs. The fitness function appears to direct the search towards generating UIOs. An empirical study using a set of real and randomly generates FSMs shows that the GA outperforms (up to 62% better) or is at least as good as a random search for UIO sequences. The results also report that most UIO sequences are very short. This suggests focusing effort in generating very short UIOs using exhaustive search techniques and relying on GA to search for the remaining UIOs.

Estimating the ease of execution for transition paths in EFSMs is one of the main problems when using FSM test case generation techniques for EFSMs. Chapter 4 defines a new method of representation for the conditions of transitions in EFSMs (PB notation). This enables the use of a fitness function of $O(l)$ complexity for a transition path of size l that is used to guide a GA search to generate FTPs. A number of pre-processes are used by the fitness function of complexity no worse than $O(|T|.log|T|)$, where $|T|$ is the number of transitions in an EFSM. The ease of execution for a TP is measured using an quality estimation algorithm, that attempts to trigger that TP a set number of times.

The fitness evaluation results suggest that the principles used in the fitness function estimate the feasibility of a given transition path (TP) with reasonable accuracy. The 0.72 and 0.62 correlation factors suggest a good correlation between the fitness algorithm and estimated feasibility for the TPs for the two EFSMs tested.

The GA search results suggest that the novel EFSM abstraction using predicate branches (PBs) could help in the search for feasible transition paths (FTPs). Compared with the alternative transition notation results, the PB notation searches generated better results for both EFSMs. It was also found that a GA that used the fitness function was more effective in FTP generation than a random algorithm.

Chapter 5 combined the approaches for generating FTPs (Chapter 4) and generating UIOs (Chapter 3) in an attempt to generate easy to trigger state verification test sequences in EFSMs. A fitness function analogous to the fitness function for generating FTPs is defined that guides a search for SVTPs. The SVTP quality estimation algorithm is also similar to the algorithm used for FTPs.

The fitness evaluation results suggest that the principles used in the fitness function estimate the combination of ease of execution and unique observable behaviour of a given TP with reasonable accuracy. Correlation factors of 0.89 and 0.56 suggest a good correlation between the fitness algorithm and estimated SVTP quality for the TPs for the two EFSMs tested.

The heuristic search results suggest that combining the search for FTPs, examined in Chapter 4, and the search for UIOs, in Chapter 3, could help in the search for SVTPs. Compared with the transition notation results, the PB notation searches generate better results for both EFSMs. GAs that used the fitness function appear to be considerably more effective in SVTP generation than a random algorithm.

Ultimately these test case generation approaches can be integrated in existing test automation and model checking tools for FSMs and EFSMs to provide a fully or partially automated methods for generating test cases with specific characteristics for large systems.

6.1 Future work

There are many topics of interest related to this work that can be explored. The principles used in this work may be applied to other NP-hard problems in FSM based testing where a robust (effective and efficient) fitness function can be defined.

In addition a specific focus for future work on FTPs and SVTPs is to improve fitness estimation for loops. The results showed that performance was negatively affected by loops in EFSMs. Some more experimental work could further help increase the accuracy of the fitness algorithms or transformations in attempt to remove or reduce the problem.

Another topic of future work is the application of heuristics to nondeterministic FSM and EFSM models. Ease or probability of execution of TPs in nondeterministic models can help in test case generation.

Another focus of future work is looking at the state verification problem in CFSMs, which is outlined in the next section.

By addressing the ease of execution problem in EFSMs, a range of additional test case generation issues besides state verification can be addressed such as test case fault resilience and configuration verification.

6.2 Testing CFSMs

A particular focus of future work is the input sequence generation for CFSMs using heuristics search like GAs. The problem of observing local transitions (of individual CFSMs) within a global transition (a combination of interlinked transitions within the set of CFSMs) is outlined in [Derd 04]. It is based on a notion of generating transition sequences with such observable properties that can identify internal transitions, given a number of conditions on the global state (the combination of states of all the CFSMs) are met. Such transition sequences, known as a Constrained Identification Sequences (CIS) were introduced by Hierons in [Hier 01]. This future work can extend this idea regarding local transitions and their corresponding final states and use genetic algorithms to automatically find test sets.

6.3 Summary

The work in this thesis presents evidence that search based techniques can be useful in automating the test case generation for FSM based specifications (including EF-SMs). Although such techniques are not guaranteed to find the optimal test cases, in terms of effectiveness and cost, they are computationally easy and require little or no human intervention. The studies in this thesis also suggest that many FSM based test sequences are very short. Hence using exhaustive generation of very short test sequences may cover much of the test effectiveness criteria. Heuristic search can be used for the remaining, more difficult to generate test cases. These techniques can be easily integrated in existing automated software testing tools and contribute to the automation of testing finite state machine based systems.

Bibliography

[Abri 96] J. Abrial. *The B Book - Assigning Programs to Meaning.* Cambridge University Press, 1996.

[Aho 91] A. Aho, A. Dahbura, D. Lee, and M. U. Uyar. "An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tour". *IEEE Transactions on Communications*, Vol. 39, pp. 1604–1615, 1991.

[Andr 86] S. J. Andriole. *Software Validation, Verification, Testing and Documentation.* Princeton, NJ: Petrocelli Books, 1986.

[Andr 88] D. Andres and P. Gibbins. *An introduction to Formal Methods of Software Development.* Milton Keyness, UK: The Open University, 1988.

[Baya 05] A. A. Bayazit and S. Malik. "Complementary use of runtime validation and model checking". In: *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pp. 1052–1059, IEEE Computer Society, Washington, DC, USA, 2005.

[Beas 93a] D. Beasley, D. R. Bull, and R. R. Martin. "An Overview of Genetic Algorithms Part 1: Fundamentals". *University Computing*, Vol. 15, pp. 58–69, 1993.

[Beas 93b] D. Beasley, D. R. Bull, and R. R. Martin. "An Overview of Genetic Algorithms Part 2: Research Topics". *University Computing*, Vol. 15, pp. 170–181, 1993.

[Beiz 90] B. Beizer. *Software testing techniques.* Van Nostard Reinhold, New York, 1990. 2nd edition.

[Beli 89] F. Belina and D. Hogrefe. "The CCITT Specification and Description Language SDL". *Computer Networks and ISDN Systems*, Vol. 16, pp. 311–341, 1989.

[Berg 89] J. A. Bergstra. *Algebraic specification.* ACM Press, New York, NY, USA, 1989.

[Boch 90] G. V. Bochmann. "Specifications of a simplified transport protocol using different formal description techniques". *Comput. Netw. ISDN Syst.*, Vol. 18, No. 5, pp. 335–377, 1990.

[Boeh 81] B. W. Boehm. *Software Engineering Economics.* Prentice Hall, 1981.

[Bott 02] L. Bottaci. "Instrumenting Programs With Flag Variables For Test Data Search By Genetic Algorithms.". In: *GECCO*, pp. 1337–1342, 2002.

[Bran 83] D. Brand and P. Zafiropulo. "On Communicating Finite-State Machines". *J. ACM*, Vol. 30, No. 2, pp. 323–342, 1983.

[Chil 94] J. Chilenski and S. Miller. "Applicability of Modified Condition/Decision Coverage to Software Testing". *Software Engineering Journal*, Vol. 9, p. 193200, 1994.

[Chow 78] T. S. Chow. "Testing software design modelled by Finite State Machines". *IEEE Transactions on Software Engineering*, Vol. 4, pp. 178–187, 1978.

[Coff 71] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". *ACM Comput. Surv.*, Vol. 3, No. 2, pp. 67–78, 1971.

[DeMi 78] R. DeMillo, R. Lipton, and F. Sayward. "Hints on test data selection: Help for the practicing programmer". *IEEE Transactions on Computers*, Vol. 12, pp. 34–41, 1978.

[Derd 02] K. Derderian. "General Genetic Algorithm Tool". Tech. Rep., www.karnig.co.uk/ga/ggat.html, 2002.

[Derd 04] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. "Input Sequence Generation for Testing of Communicating Finite State Machines CFSMs.". In: *LNCS vol. 3103*, pp. 1429–1430, Springer, 2004.

[Derd 05] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. "Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms". In: *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pp. 1081–1082, ACM Press, New York, NY, USA, 2005.

[Derd 06] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. "Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs". *The Computer Journal*, Vol. 49, No. 3, pp. 331–344, 2006.

[Derr 99] J. Derrick and E. Boiten. "Testing Refinements of State-based Formal Specifications". *Software Testing, Verification and Reliability*, No. 9, pp. 27–50, July 1999.

[Dijk 59] E. W. Dijkstra. "A note on two problems in connexion with graphs.". *Numerische Mathematik*, Vol. 1, pp. 269–271, 1959.

[Dual 00] A. Y. Duale and M. Ü. Uyar. "Generation of Feasible Test Sequences for EFSM Models". In: *TestCom '00: Proceedings of the IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems*, p. 91, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2000.

[Dual 04] A. Y. Duale and M. Ü. Uyar. "A Method Enabling Feasible Conformance Test Sequence Generation for EFSM Models.". *IEEE Transactions Computers*, Vol. 53, No. 5, pp. 614–627, 2004.

[Gibb 85] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1985.

[Gogu 88] J. Goguen and T. Walker. *Introducing OBJ3.* Computer Science Laboratory SRI Internationl Report SRI-CSL-88-9, 1988.

[Gold 89] D. E. Goldberg. *Genetic Algorithms in search, optimisation and machine learning.* Addison-Wesley Publishing Company, Reading, Mass. USA, 1989.

[Gold 93] D. Goldberg, K. Deb, and D. Theirens. "Toward a better understanding of mixing in genetic algorithms". *Society of Instrument and Control Engineers Journal*, Vol. 32, pp. 10–16, 1993.

[Gone 70] G. Gonenc. "A method for the design of fault detection experiments". *IEEE Transactions on Computers*, Vol. 19, pp. 551–558, 1970.

[Guo 04] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. "Computing Unique Input/Output Sequences using Genetic Algorithms". In: *Formal Approaches to Software Testing: Third International Workshop, FATES 2003, LNCS vol. 2931*, pp. 169–184, Springer, New York, 2004.

[Halc 98] M. Halcombe and F. Ipate. *Correct Systems: Building a Business Process Solution.* Springer, 1998.

[Hare 98] D. Harel and M. Politi. *Modeling reactive systems with statecharts: the STATEMATE approach.* McGraw-Hill, 1998.

[Henn 64] F. C. Hennie. “Fault–detecting experiments for sequential circuits”. In: *Proceedings of Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pp. 95–110, Princeton, New Jersey, November 1964.

[Hier 01] R. M. Hierons. “Checking states and transitions of a set of communicating finite state machines”. *Microprocessors and Microsystems,Special Issue on Testing and testing techniques for real-time embedded software systems*, Vol. 24, pp. 443–452, 2001.

[Hier 02] R. M. Hierons and H. Ural. “Reduced Length Checking Sequences”. *IEEE Transactions on Computers*, Vol. 51, No. 9, pp. 1111–1117, 2002.

[Hier 03] R. M. Hierons, T. H. Kim, and H. Ural. “On The Testability of SDL Specifications”. *Computer Networks*, Vol. 44, pp. 681–700, 2003.

[Hier 04] R. M. Hierons. “Testing from a Non–Deterministic Finite State Machine Using Adaptive State Counting”. *IEEE Transactions on Computers*, Vol. 53, No. 10, pp. 1330–1342, 2004.

[Hoar 85] A. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[Hogr 91] D. Hogrefe. “OSI formal specification case study: the Inres protocol and service.”. Tech. Rep. 5, University of Bern, IAM-91-012 1991.

[Holz 90] G. J. Holzmann. *Design and Validation of Protocols.* Prentice-Hall, 1990.

[Hopc 79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[Hwan 01] I. Hwang, T. Kim, S. Hong, and J. Lee. “Test selection for a nondeterministic FSM”. *Computer Communications*, Vol. 24, No. 12, pp. 1213–1223, 2001.

[Inan 99] K. Inan and H. Ural. “Efficient checking sequences for testing finite state machines”. *Information and Software Technology*, Vol. 41, pp. 799–812, 1999.

[Jone 90] C. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[Jone 96] B. F. Jones, H.-H. Sthamer, and D. E. Eyres. “Automatic structural testing using genetic algorithms”. *The Software Engineering Journal*, Vol. 11, No. 5, pp. 299–306, 1996.

[Jone 98] 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*, Vol. 41, No. 2, pp. 98–107, 1998.

[Koha 78] Z. Kohavi. *Switching and finite automata theory*. McGraw-Hill, New York, 1978.

[Lee 94] D. Lee and M. Yannakakis. "Testing Finite State Machines: State Identification and Verification". *IEEE Transactions on Computers*, Vol. 43, pp. 306–320, 1994.

[Lee 96] D. Lee and M. Yannakakis. "Principles and methods of testing finite state machines - a survey". *Proceedings of the IEEE*, Vol. 84, pp. 1090–1123, 1996.

[LGSy 91] LGSynth91. "Logic synthesis and optimization benchmarks". Tech. Rep. 3, University of California, 1991. www.ece.pdx.edu/polo/function/LGSynth91.

[Li 94] X. Li, T. Higashino, M. Higuchi, and K. Taniguchi. "Automatic Generation of Extended UIO Sequences for Communication Protocols in EFSM Model". In: *Distributed Processing System No.066*, pp. 225–240, IPSJ SIGNotes, Japan, November 1994.

[Luo 94a] G. Luo, G. von Bochmann, and A. Petrenko. "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized WP-Method". *IEEE Transactions on Software Engineering*, Vol. 20, No. 2, pp. 149–162, Feb. 1994.

[Luo 94b] G. Luo, G. von Bochmann, and A. Petrenko. "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method". *IEEE Transactions on Software Engineering*, Vol. 20, No. 2, pp. 149–162, 1994.

[McMi 04] P. McMinn. "Search-based software test data generation: a survey.". *Softw. Test., Verif. Reliab.*, Vol. 14, No. 2, pp. 105–156, 2004.

[Mich 01] C. C. Michael, G. McGraw, and M. A. Schatz. "Generating Software Test Data by Evolution". *IEEE Transactions on Software Engineering*, Vol. 27, No. 12, pp. 1085–1110, 2001.

[Mich 96] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag Berlinn Heidelberg New York, 1996. Third, Revised and Extended Edition.

[Mill 93] R. E. Miller and S. Paul. "On the generation of minimal-length conformance tests for communication protocols". *IEEE/ACM Trans. Netw.*, Vol. 1, No. 1, pp. 116–129, 1993.

[Miln 89] R. Milner. *Communication and Concurency.* Prentice Hall, 1989.

[Moor 56] E. Moore. "Gedanken-experiments on sequential machines". *Automata studies*, Vol. 34, pp. 129–153, 1956.

[Moss 04] P. D. Mosses. *CASL Reference Manual, The Complete Documentation of the Common Algebraic Specification Language.* Vol. 2960 of *Lecture Notes in Computer Science*, Springer, 2004.

[Naik 95] K. Naik. "Fault-tolerant UIO sequences in finite state machines". In: *8th IFIP International Workshop on Protocol Test Systems*, pp. 201–214, Evry, France, September 1995.

[Ntaf 88] S. Ntafos. "A Comparison of Some Structural Testing Strategies". *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp. 868–874, 1988.

[Ostr 88] T. J. Ostrand and M. J. Balcer. "The category-partition method for specifying and generating fuctional tests". *Commun. ACM*, Vol. 31, No. 6, pp. 676–686, 1988.

[Parg 99] R. P. Pargas, M. J. Harrold, and R. R. Peck. "Test–data generation using genetic algorithms". *Journal of Software Testing Verification and Reliability*, Vol. 9, No. 4, pp. 263–282, 1999.

[Pear 79] S. W. Pearson and J. E. Bailey. "Measurement of Computer User Satisfaction.". In: *Int. CMG Conference*, pp. 49–58, 1979.

[Petr 04] A. Petrenko, S. Boroday, and R. Gorz. "Confirming Configurations in EFSMs". *IEEE Transactions on Software Engineering*, Vol. 30, pp. 29–42, January 2004.

[Petr 96] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. "Testing deterministic implementations from nondeterministic FSM specifications". In: *IFIP TC6 9th International Workshop on Testing of Communicating Systems*, pp. 125–140, Chapman & Hall, Ltd., Darmstadt, Germany, 9–11 September 1996.

[Rama 03] T. Ramalingom, K. Thulasiraman, and A. Das. "Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines.". *Computer Communications*, Vol. 26, No. 14, pp. 1622–1633, 2003.

[Rapp 85] S. Rapps and E. J. Weyuker. “Selecting software test data using data flow information”. *IEEE Trans. Softw. Eng.*, Vol. 11, No. 4, pp. 367–375, 1985.

[Reza 95] A. Rezaki and H. Ural. “Construction of checking sequences based on characterization sets”. *Computer Communications*, Vol. 18, No. 12, pp. 911–920, 1995.

[Sabn 88] K. K. Sabnani and A. T. Dahbura. “A protocol test generation procedure”. *Computer Networks and ISDN Systems*, Vol. 15, pp. 285–297, 1988.

[Shen 89] Y. Shen, F. Lombardi, and T. Dahbura. “Protocol conformance testing using multiple UIO sequences”. In: *IFIP WG6.1 9th Int. Symp. on Protocol Specification Testing and Verification*, pp. 131–144, Amsterdam, Holland, 1989.

[Shen 91] X. Shen, S. Scoggins, and A. Tang. “An improved RCP-method for protocol test generation using backward UIO sequences”. In: *ACM Symposium on Applied Computing (SAC 1991)*, pp. 284–293, ACM Press New York, Kansas City, MO, USA, April 1991.

[Shen 92] X. Shen and G. Li. “A new protocol conformance test generation method and experimental results”. In: *SAC ’92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, pp. 75–84, ACM Press, 1992.

[Sidh 89] D. P. Sidhu and T. K. Leung. “Formal Methods for Protocol Testing: A Detailed Study”. *IEEE Transactions on Software Engineering*, Vol. 15, pp. 413–426, 1989.

[Son 98] H. Son, D. Nyang, S. Lim, J. Park, Y.-H. Choe, B. Chin, and J. Song. “An Optimized Test Sequence Satisfying the Completeness Criteria”. In: *12th International Conference on Information Networking (ICOIN-12)*, pp. 621 – 625, IEEE, Tokyo, Japan, January 1998.

[Spiv 88] J. Spivey. *Understanding Z: A specification language and its formal smantics.* Cambridge University Press, 1988.

[Spiv 89] J. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1989.

[Srin 94] M. Srinivas and L. M. Patnaik. “Genetic Algorithms: A Survey”. *IEEE Computer*, Vol. 27, pp. 17–27, 1994.

[Stan 88] I. O. for Standartization (ISO). "Information processing systems - Open systems interconnections - LOTOS - A formal description technique based on the temporal ordering of observable behaviour". *ISO8807*, Vol. , p. , 1988.

[Sun 01] H. Sun, M. Gao, and A. Liang. "Study on UIO sequence generation for sequential machine's functional test". In: *4th International Conference on ASIC,OCT 23-25, 2001*, pp. 628–632, IEEE, Shanghai, China, October 2001.

[Sun 98] D. Sun, B. Vinnakota, and W. Jiang. "Fast state verification". In: *Proceedings of the 35th annual conference on Design automation*, pp. 619 – 624, ACM Press New York, San Francisco, California, United States, June 1998.

[Tane 96] A. S. Tanenbaum. *Computer Networks.* Prentice Hall, Upper Saddle River, NJ, USA, 1996. 3rd edition.

[Trac 00] N. Tracey, J. Clark, K. Mander, and J. McDermid. "Automated test-data generation for exception conditions". *Software Practice and Experience*, Vol. 30, No. 1, pp. 61–79, 2000.

[Trak 73] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite Automata: Behavior and Synthesis.* North-Holland, Amsterdam, 1973.

[Ural 97] H. Ural, X. Wu, and F. Zhang. "On Minimizing the Lengths of Checking Sequences". *IEEE Transactions on Computers*, Vol. 46, No. 1, pp. 93–99, 1997.

[Wang 87] B. Wang and D. Huthinson. "Protocol testing techniques". *Computer communications*, Vol. 10, pp. 79–87, 1987.

[Wegn 97] J. Wegner, H. Sthamber, B. Jones, and D. Eyres. "Testing Real-time systems using Genetic Algorithms". *Software Quality*, Vol. 6, pp. 127–135, 1997.

[Whit 99] D. Whitley. "A Free Lunch Proof for Gray versus Binary Encodings". In: *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 726–733, Morgan Kaufmann, CA, USA, Orlando, Florida, USA, July 1999.

[Yang 90] B. Yang and H. Ural. "Protocol conformance test generation using multiple UIO sequences with overlapping". In: *SIGCOMM '90: Proceedings* *of the ACM symposium on Communications architectures & protocols*, pp. 118–125, 1990.