An Integrated Search-Based Approach for Automatic Testing from Extended Finite State Machine (EFSM) Models

Abdul Salam Kalaji, Robert Mark Hierons and Stephen Swift
School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, UB8 3PH, UK
e-mail: {abdulsalam.kalaji, rob.hierons, stephen.swift}@brunel.ac.uk

Abstract

Context: The extended finite state machine (EFSM) is a modelling approach that has been used to represent a wide range of systems. When testing from an EFSM, it is normal to use a test criterion such as transition coverage. Such test criteria are often expressed in terms of transition paths (TPs) through an EFSM. Despite the popularity of EFSMs, testing from an EFSM is difficult for two main reasons: path feasibility and path input sequence generation. The path feasibility problem concerns generating paths that are feasible whereas the path input sequence generation problem is to find an input sequence that can traverse a feasible path. Objective: While search-based approaches have been used in test automation, there has been relatively little work that uses them when testing from an EFSM. In this paper, we propose an integrated search-based approach to automate testing from an EFSM. Method: The approach has two phases, the aim of the first phase being to produce a feasible TP (FTP) while the second phase searches for an input sequence to trigger this TP. The first phase uses a Genetic Algorithm whose fitness function is a TP feasibility metric based on dataflow dependence. The second phase uses a Genetic Algorithm whose fitness function is based on a combination of a branch distance function and approach level. Results: Experimental results using five EFSMs found the first phase to be effective in generating FTPs with a success rate of approximately 96.6%. Furthermore, the proposed input sequence generator could trigger all the generated feasible TPs (success rate = 100%). Conclusion: The results derived from the experiment demonstrate that the proposed approach is effective in automating testing from an EFSM.

***Keywords*: Search-based testing, EFSM, feasible transition paths, automatic test derivation.**

1. Introduction

Faults in systems can have severe consequences and therefore it is generally accepted that testing is a vital stage of software development. In software testing we supply the implementation under test (IUT) with a test case that consists of a finite sequence of inputs and outputs and then observe the resultant output and compare it to that stated in the test case. Testing can constitute up to 50% of the overall software development cost [1, 2]. Therefore, it is natural for the software engineering community to try to develop methods to reduce the cost associated with testing while enhancing the testing process. Since manual testing is expensive, time consuming and error prone, there has been significant interest in automation [3], [4], [5], [6].

Most approaches to testing can be categorised as white-box or black-box. In white-box testing, a tester has access to the internal structure of a system (such as the source code and algorithms) and uses this knowledge to derive test cases. However, it is often the case that we do not have access to the source code of the system since, for example, it may contain components that were outsourced. In contrast, black-box testing does not use information about the internal system structure and so it is normal for the tester to use the system specification (usually represented as a model) to derive the test cases. Then these test cases are applied to the IUT and the resultant outputs are monitored. By comparing the produced outputs to those stated in the specification, the tester can determine whether the IUT conforms to the specification on that test case. Therefore, such testing is commonly referred to as specification-based testing.

In order to apply specification-based testing, it is necessary to have a system specification, usually represented in terms of a model. Two modelling approaches that can be used for this purpose are finite state machines (FSMs) and extended finite state machines (EFSMs) [7]. An FSM comprises of a finite set of states and transitions among the states. Each transition has a start state and an end state. Also, a transition requires an input and it produces an output. The FSM is used to model a system that has a control part, for example a basic telephone device. However, if the system has, in addition to the control part, a data part, then an extended FSM can be used. An EFSM can model both control and data parts since it extends the FSM structure with a set of variables (memory). Therefore, in an EFSM, a transition can have guards (preconditions) over the inputs and the machine's variables and also can have operations (assignments) to these variables. EFSMs can model both control and data and a number of standard modelling languages such as Statecharts and SDL can be seen as EFSMs. EFSM based test techniques can be applied to models in such languages [7, 8] as well as formalisms such as (UML) Use Cases and Z and in domains such as communication protocols and web services [8], [9], [10], [11], [12], [13], [14].

When testing from an EFSM, a test case is required and this consists of a sequence of inputs and outputs. In this paper we focus on deriving the input section of a test case. The input section of a test case is a sequence of inputs where each input can also have parameters. We might produce test cases with the aim of achieving a coverage criterion such as transition coverage or state coverage [15]. Transition coverage, for example, requires that the test cases, between them, lead to all transitions of the EFSM being exercised. Each test case defines a transition path (TP) through the EFSM: the path traversed when the EFSM is stimulated with the input sequence from the test case. Many coverage criteria can be seen as requirements on the TPs defined by the test cases used. For example, transition coverage requires that each transition of the EFSM is contained in a TP that corresponds to one of the test cases. Therefore, when testing from an EFSM, we might derive a set of TPs that satisfies the given test criterion and then produce test cases to trigger these TPs. However, since an EFSM's transitions may have guards (preconditions) and operations, a given TP may be infeasible. For example, one transition's operation may assign the value 0 to a variable x while a later transition's guard requires $x>0$ despite the value of x not having changed between these transitions. Such a path is infeasible and so it is impossible to find an input sequence (test case) to trigger it. However, the problem of determining whether a given path is feasible is undecidable and

the development of good methods is an open research problem [16, 17]. If the path is feasible, then an input sequence is required to trigger (exercise) this path but it can be difficult to find such an input sequence since the input domain is usually large but the required input values might constitute just a small subset of this domain [18]. For example, a machine variable *x* can be of integer data type, but the required values to exercise a guard over *x* can be within a tiny range.

While model-checkers can be used to generate test cases from EFSMs, the presence of several internal variables with a large range of values (e.g. integers) can lead to a state space explosion. This paper instead explores the use of search-based techniques in testing from an EFSM. It is to be expected that there are situations in which model-checkers outperform search-based methods but also situations in which search-based methods outperform model-checkers. For example, a study by Nilsson et al. [19] found that when testing from dynamic systems, the search problem became more difficult and a Genetic Algorithm (GA) search outperformed a model checker. Also, a recent study by Wenzel et al. [20] states that using model checkers to generate test cases is more expensive than using heuristic techniques. Characterising situations in which one approach is superior to the other is an important problem for future work.

Although optimisation algorithms have proven efficient for automating aspects of testing [5], very little attention has been paid towards investigating their application to EFSM testing. This paper proposes a novel search-based approach to automate testing from EFSMs. The proposed approach has two components. The first uses a TP feasibility metric, which is a metric that aims to reward TPs that are 'likely to be feasible'. The feasibility metric is based on an analysis of the dataflow dependence among the operations and guards in a path's transitions. This feasibility metric is used to guide a search for TPs with the aim of finding feasible TPs that satisfy the test criterion. Once a set of TPs is obtained, in the second part a fitness function guides the search for an input sequence that can trigger a given TP.

The main contributions of this paper are the following:

1. It describes a search-based method that directs the automatic generation of TPs from an EFSM model with the intention that the resultant TPs are feasible.
2. It proposes a search-based method for automatically generating input sequences that can trigger a given feasible transition path (FTP).
3. The paper is the first to propose an integrated search-based approach for automatic testing from an EFSM.
4. The paper, also, statistically investigates the relationship between the proposed TP feasibility metric and the effort in terms of time that is required to generate an input sequence to exercise the TP.
5. The paper empirically validates the proposed approach by using it with five EFSMs.

The feasibility metric was devised with the aim of directing search towards paths that are feasible. Part of the motivation is that, when testing using a given test criterion, the feasibility metric can be combined with another metric that directs the search towards test cases that contribute to the satisfaction of the test criterion. For example, if we wish to find test cases that cover every state of the EFSM then for a state *s* we could devise a metric that estimates how close a test case is to passing through *s*. The two metrics can then be

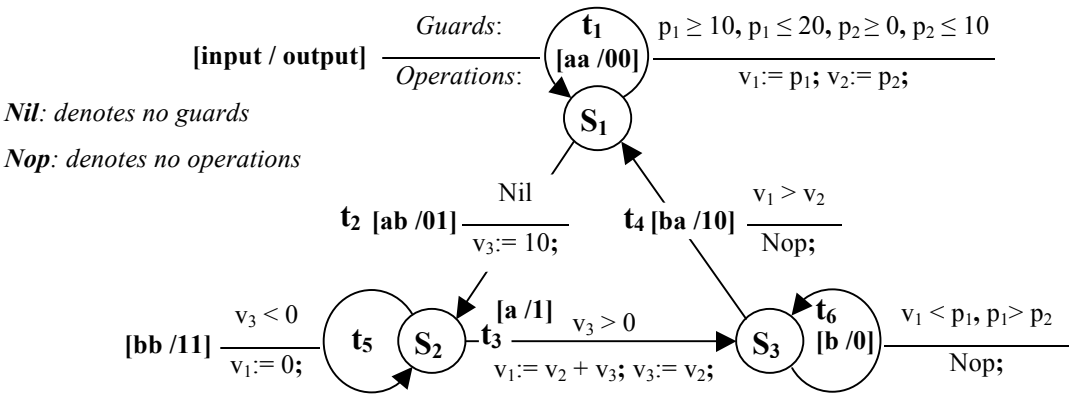


Figure 1. An EFSM example (*M*)

used to search for paths that are likely to be feasible and pass through *s*. The hope also was that paths with a good (low) fitness according to our feasibility metric will be relatively easy to trigger using test generation methods based on search. The results of experiments reported in Section 5 suggest that the proposed feasibility metric does achieve this: in the experiments there was a statistically significant strong correlation between the feasibility metric value for a path and the number of generations required by a GA to find an input sequence to follow that path. Interestingly, a similar result was found when a constraint solver was used to generate input sequences instead of a GA.

The rest of the paper proceeds as follows: Section 2 provides background information, including a description of search-based testing. In Section 3, the proposed approach is described. Experimental results are provided in Sections 4 and 5 and related work is described in Section 6. Concluding remarks and future work are in Section 7.

2. Preliminaries

2.1. The Extended Finite State Machine (EFSM)

A finite state machine (FSM) is a Mealy machine [21] (or transducer), which has finite sets of states, inputs, and outputs. An output is produced upon state transition and this occurs when applying an input to the machine. When extending a Mealy machine with context variables, predicates, and operations we get an extended finite state machine (EFSM). An EFSM is a 6-tuple [18] (S, s_0, V, I, O, T) where: S is the finite set of logical states, s_0 is the initial state, V is the finite set of (internal) context variables, I is the finite set of input declarations, O is the finite set of output declarations, and T is the finite set of transitions. The transition $t \in T$ is represented by the 5-tuple (s_s, i, g, op, s_e) in which: s_s is the start state of t, i is the input where $i \in I$ and i may have associated input parameters, g is a logical expression called the guard, op is the sequential operation which consists of simple statements such as output statements and assignment statements, and s_e is the end state of t.

In an EFSM, a state transition occurs when one of the machine's transitions is taken. If the state is s_s then transition $t = (s_s, i, g, op, s_e)$ can be taken if input i is received and the guard g is satisfied. If this happens then the operations in op are executed and the logical state becomes s_e. Both g and op can refer to input parameters and context variables. An EFSM is deterministic if for any group of transitions with the same input that leave a state, it is not possible to satisfy the guards of more than one transition in this group at the same time [22]. In this paper, we only consider deterministic EFSMs.

The simple EFSM M shown in Fig. 1 will be used throughout this paper to aid the description of the proposed approach. M has three states, six transitions and three context variables v_1, v_2 and v_3. If, for example, M is at state S_1 then in order to fire t_1 an input *aa* is required together with two input parameters p_1 and p_2. If the values of p_1 and p_2 satisfy the guard of t_1 ($p_1 \geq 10$, $p_1 \leq 20$, $p_2 \geq 0$, $p_2 \leq 10$) then t_1 is fired, the operations ($v_1 := p_1$**;** $v_2 := p_2$;) are executed and the machine outputs 00. These operations update the values of v_1 and v_2. Since t_1 ends at the same state, S_1, then the machine remains at this state. In testing from an EFSM we supply an input sequence along with parameter values and each such sequence defines a TP.

2.2. Program Data Flow Dependence

Given a program and a variable x within this program, a statement at which x appears can be an assignment to x or a use of x (or both). An assignment to x defines or updates the value of x and so x is said to be *defined* at such a statement. A use of x occurs when x is referenced in a predicate (a predicate use/p-use) or x is referenced in a computation that either updates the value of a variable or is produced as output (a computation use/c-use). Give a path between two statements n_1 and n_2, if x is not defined after n_1 and before n_2 then the path from n_1 to n_2 is a *definition clear path* for x [23]. If, in addition, n_1 is a definition of x and n_2 is a use of x, then statements n_1 and n_2 form a definition-use (*du*) pair for x and there is dataflow dependence between n_1 and n_2 [24].

Consider a transition path (TP) through M which consists of t_2t_3 (Fig. 1), the context variable v_3 is defined by t_2, then used by the guard of t_3 (p-use) and the path is definition clear for v_3. Thus, t_2t_3 forms a *du* pair for v_3 and there is dataflow dependence between the two transitions. In this paper we utilise dataflow information in EFSMs to define the proposed TP feasibility metric.

2.3. Symbolic Execution

Symbolic execution is an analysis approach which allows a program to be executed using a set of symbolic inputs [25]. The execution here is similar to normal program execution. However, the inputs are given in terms of symbols, and thus the outputs of the program are symbols and expressions over these symbols. This is particularly useful to understand the relationship between a given input and its associated output.

When using symbolic execution, the problem of test case generation can be reformulated to the problem of solving a set of algebraic expressions that result from symbolically executing a selected path. For example, consider the path t_1t_1 in the machine M (Fig. 1). Let the parameters used to trigger the first t_1 be p_1 and p_2 and the parameters used to trigger the next t_1 be p_3 and p_4. Thus we have the following constraints: ($p_1 \geq 10$ AND $p_1 \leq 20$ AND $p_2 \geq 0$ AND $p_2 \leq 10$ AND $p_3 \geq 10$ AND $p_3 \leq 20$ AND $p_4 \geq 0$ AND $p_4 \leq 10$). If these constraints can be solved (e.g. $p_1 = 10$; $p_2 = 5$; $p_3 = 20$; $p_4 = 9$) then the solution defines an input sequence that can exercise the considered path.

In Section 5, we used a constraint-based testing (CBT) approach as an alternative method to generate input sequences to trigger the generated feasible transition paths.

2.4. Search-Based Testing

It has been observed that many software engineering problems can be expressed as search problems over sets of complex entities and this has led to interest in search-based software

engineering [26]. In search-based software engineering, typically problems are expressed as optimisation problems and metaheuristic techniques such as hill climbing, simulated annealing and genetic algorithms are applied in order to find acceptable solutions [27].

Search-based software testing (SBST) is an approach that reformulates software testing problems as optimisation problems. This reformulation serves the purpose of allowing the automatic derivation of test cases that satisfy a given test criterion. In SBST we need a representation of the candidate solutions and a method, called a *fitness function*, which can evaluate the candidate solutions.

Solution representation is a method that allows a search technique to manipulate candidate solutions. There are many methods that can be used to represent candidate solutions. If the candidate solutions are numbers, then it is possible to use binary valued encoding, integer valued encoding or real valued encoding. In binary encoding, for example, each number is represented by its equivalent binary value. For example, (7, 8) can be represented as (0111, 1000). Often different forms of encoding can be used depending on the input domain of the problem. For example, if the inputs are integers then integer valued representation is possible. Similarly, real valued encoding is an option when inputs are real numbers.

A fitness function is required to compare candidate solutions. The fitness function measures how good each candidate solution is. The fitness function assigns each candidate solution a positive number that estimates how far it is from being an acceptable solution. Since the optimisation problem is a minimisation one, candidate solutions that receive lower fitness values are better and acceptable solutions usually have a fitness value of zero.

If we have a representation and a fitness function then we can apply a metaheuristic search technique. Genetic algorithms are a well-know metaheuristic technique that have been widely applied to white-box testing (for examples see [5]). The next subsection describes the basic genetic algorithm.

2.5. Genetic Algorithms

Genetic algorithms (GAs) are metaheuristic search techniques that have been found to be powerful, simple, and robust [28]. In order to apply a GA to an optimisation problem, a solution representation (encoding) is required. When solutions are encoded, each is called a *chromosome* and consists of components that are called *genes*. For example, let the initial set of solutions be integer values such as {7, 6, 8}. If binary encoding is used, then {0111, 0110, 1000} represents the chromosomes (individuals). Any bit of a chromosome represents a gene with a value of either 0 or 1.

The GA cycle consists of the main operators: *evaluation*, *selection, crossover*, and *mutation*. The GA cycle starts by evaluating the fitness of each individual which is a positive value that measures how 'fit' this individual is and influences its chance of being selected as a parent. Evaluating the fitness of each individual is performed through calling a fitness function which is problem dependent.

Deriving an effective fitness function for a given problem is a central task when applying a search technique such as a GA. For some problems, a fitness function can be derived directly. For example, consider minimising the function:

$$f(x) = x^2 \text{ ; where } 0 \leq x \leq 10^3$$

For such a problem, a fitness function can be similar to the function that describes the

problem (i.e. fitness(x) = x^2) or with a slight difference (i.e. fitness(x) = x). Such a fitness function can differentiate candidate solutions and prioritises those that have better values. However, this is not always possible and then alternative metrics are required [29]. For example, consider the metric that measures statement coverage of source code. This metric can be utilised as a fitness function to measure the number of statements that are covered by a given test case (candidate solution).

After individuals are evaluated, a *selection* based on fitness is made to perform 'breeding'. There are many selection methods, such as roulette wheel and ranking, that can be used [30]. Through breeding new individuals are introduced. This is accomplished by applying a *crossover* operator that acts on two individuals to produce two new individuals. There are several approaches to crossover including one-point crossover, which operates by choosing a random position on the individual's bit string, and then the substrings before that position are kept while the tails are swapped. For example, crossover on the two parents $Parent_1$ and $Parent_2$ before position 4, yields the offspring $Child_1$ and $Child_2$.

$Parent_1$ {011|00} ⟹ $Child_1$ {011|11}

$Parent_2$ {101|11} ⟹ $Child_2$ {101|00}

In order to maintain population diversity, new characteristics are infrequently injected by applying *mutation*. Mutation acts on one individual at a time and randomly changes the values of some of the individual's genes [31]. For example, $Child_1$ might become $Child_1'$ after mutating the bits on positions 1 and 5.

$Child_1$ {01111} ⟹ $Child_1'$ {11110}

These operators yield new individuals that either replace the old generation (population) or a selection is used to obtain a new population from the previous population and the new individuals. The population undergoes a number of updates until satisfying one of the stopping criteria such as finding a satisfactory solution or reaching a maximum number of generations [30].

3. The Proposed Approach

The proposed approach has two phases. The first phase generates TPs to satisfy a given test criterion and in this paper we use transition coverage; in principle the method can be extended to other criteria. For each transition t, a GA is used to find a TP that executes t. The fitness function of the GA is a feasibility metric, the aim being that this guides the search towards TPs that are likely to be feasible TPs (FTPs). The method described in this paper thus uses a GA to find TPs with good values for the feasibility metric and chooses one such TP that includes t. However, it is likely that the method can be improved by including additional information that guides the GA towards paths that contain t and this will be important when it is hard to find paths that contain t.

The proposed feasibility metric is mainly based on dataflow dependencies among the transitions of a TP, where there is dependence between transitions t and t' (with t being before t' in the TP) if there is a context variable v that is assigned a value in t, referenced by the guard of t', and not assigned to between t and t'. We refer to such a pair (t, t') as an (*affecting*, *affected-by*) pair. The feasibility metric also considers guards that are not affected by any transitions, such a guard involving comparisons among input parameters and possibly constants. When evaluating a TP, all (*affecting*, *affected-by*) pairs in this TP are found and a penalty value is assigned to each pair, with the penalty depending on the

relevant assignment and guard. Guards that are included in this TP and are not affected by transitions are also penalised. In defining the penalties, assignments and guards were classified and for each possible combination of guard and assignment type, a pre-determined penalty value assigned. The use of pre-determined penalties has the benefit of making the feasibility metric evaluation relatively quick. The penalties and feasibility metric are given in Section 3.1.

Once TPs have been generated, these need to be fired (executed). The second phase of the proposed approach uses a GA, based on approach level and branch distance, to generate test inputs to fire the TPs produced by the first phase. If we fail to execute a TP we might then return to the first phase to produce an alternative TP. For a TP, we have a sequence of functions instead of one function. This is similar to the structure of nested IF statements. As a result, we first apply the fitness calculation proposed by Wegener et al. [34] to each transition in a TP and then consider each function (transition) as an IF statement. If the function (transition) is executed then it returns zero otherwise it returns a value that states how close the inputs were to executing this function. The fitness function, for generating inputs for a TP, is described in detail in Section 3.2.

3.1. Phase 1: Feasible Transition Path (FTP) Generation

Before providing a detailed description of the FTP generation method, we introduce the following definitions:

Definition 1: A *transition path* (TP) of length n through an EFSM is a sequence of n consecutive transitions $t_1, t_2, .., t_n$ that starts at the initial state of the EFSM.

Definition 2: A TP $t_1, t_2, .., t_n$ is feasible (an *FTP*) if it is possible to trigger each transition t_i, where $1 \leq i \leq n$, in the order that it appears in this TP.

Any path from the initial state of an EFSM defines a TP but only some of these paths may be FTPs. For example, the TP $t_1t_2t_3$ through the machine M shown in Fig. 1 is an FTP but $t_1t_2t_5$ is not since t_2 assigns 10 to v_3 and t_5 requires v_3 to be less than zero.

Any transition can have guards and operations. We assume that a guard consists of atomic guards combined using AND and OR. In this section we consider only atomic guards and in Section 3.1.1. we show how more general guards are treated. A guard has the form of (e *gop* e') where e and e' are expressions and *gop* $\in \{>, <, \geq, \leq, =, \neq\}$ is the guard operator.

Given expression e, we let *Ref*(e) denote the set of variables that appear in e. According to e and e' a transition's guard can be classified into the following types:

1. $\boldsymbol{g^{pv}}$: a comparison involving a parameter and one or more context variables. More formally, the guard (e *gop* e') is such that *Ref*(e) $\cup$ *Ref*(e') contains at least one input parameter and at least one context variable and also e and e' are not constants. Transition t_6 in M (Fig. 1) is an example since it inputs p_1 and then compares this with the variable v_1.
2. $\boldsymbol{g^{vv}}$: a comparison among context variables' values. More formally, the guard (e *gop* e') is such that *Ref*(e) $\cup$ *Ref*(e') contains only context variables and e and e' are not constants. Transition t_4 in M (Fig. 1) is an example where the guard compares the values of v_1 and v_2.
3. $\boldsymbol{g^{vc}}$: a comparison between a constant and an expression involving context variables. More formally, the guard (e *gop* e') is such that *Ref*(e) $\cup$ *Ref*(e') contains at least one context variable and either e or e' is a constant. An example is

the transition t_3 in M (Fig. 1) since its guard references v_3, compares it to a constant and does not reference an input parameter.

4. $\boldsymbol{g^{pc}}$: a comparison between a constant and an expression involving a parameter. More formally, the guard (e *gop* e') is such that *Ref*(e) ∪ *Ref*(e') contains at least one input parameter and either e or e' is a constant. Transition t_1 in M (Fig. 1) is an example since it compares the input p_1 to a constant.
5. $\boldsymbol{g^{pp}}$: a comparison between expressions involving parameters. More formally, the guard (e *gop* e') is such that *Ref*(e) ∪ *Ref*(e') contains at least one input parameter and no context variables and e and e' are not constants. An example is transition t_6 in M (Fig. 1) since it inputs p_1 and p_2 and then the guard involves a comparison between p_1 and p_2.

An assignment in a transition t has the form of $v := e$, where v is a context variable and e is an expression. An assignment to context variable v can be classified as one of the following:

1. $\boldsymbol{op^{vp}}$: it assigns to v a value that depends on the parameter; *Ref*(e) contains at least one input parameter. An example is the transition t_1 in M (Fig. 1) since it has an operation that assigns a parameter value to v_1.
2. $\boldsymbol{op^{vv}}$: it assigns to v a value that depends on the context variable(s); *Ref*(e) contains only context variables and e is not a constant. An example is the transition t_3 in M (Fig. 1) since it assigns the sum of the values of v_2 and v_3 to context variable v_1.
3. $\boldsymbol{op^{vc}}$: it assigns to v a constant. An example is the transition t_2 in M (Fig. 1).

Based on the classifications of guards and assignments, two types of transitions can be distinguished: affecting and affected-by transitions.

Definition 3: In a TP $t_1, t_2, .., t_n$, a transition t_i is *affecting* if t_i has an assignment $op \in \{op^{vp}, op^{vc}, op^{vv}\}$ to v and there exists a guarded transition $t_j \in$ TP, where $1 \leq i < j \leq n$, t_j has a guard $g \in \{ g^{pv}, g^{vv}, g^{vc}\}$ over v and the path between t_i and t_j is definition clear with respect to v. t_j is also said to be an *affected-by* transition.

For example, in the EFSM M (Fig. 1) the transition t_2 assigns a value to v_3 and the guard of t_3 references this variable. Furthermore, t_2t_3 is a definition clear path and so for the subsequence t_2t_3, the transition t_2 is an affecting one whereas t_3 is an affected-by transition.

Sometimes we can immediately determine that a path is infeasible and so we give the path a high (poor) fitness. The following definitions give the rules used in this work to say when a path is definitely infeasible. Naturally, additional such cases can be identified.

Definition 4: In a TP $t_1, t_2, .., t_n$, the assignment $op \in op^{vc}$ in t_i is *opposed* to the guard $g \in g^{vc}$ of t_j ($1 \leq i < j \leq n$) if there exists a variable v such that op is an assignment to v, g references v, $t_{i+1}, \ldots, t_j$ is a definition clear path for v and either the constants that appear in op and g are the same and $gop \in \{<, >, \neq\}$ or are different and $gop \in \{=\}$.

Consider again the EFSM M (Fig. 1), the assignment to v_3 in transition t_2 is opposed to the guard in t_5 since t_2 defines v_3 to be 10 and t_5 requires v_3 to be less than zero. As a result, any path that contains the subsequence t_2t_5 must be infeasible.

Definition 5: In a TP $t_1, t_2, .., t_n$, the guards $g_i, g_j \in g^{vc}$ of t_i and t_j respectively ($1 \leq i < j \leq n$) are *opposed* when there exists a variable v such that both guards reference v, the path from t_i to t_j is definition clear for v and one of the following holds:

1. The constants that appear in g_i and g_j are the same and (one $gop \in \{\neq, >, <\}$ and the other $gop \in \{=\}$ or one $gop \in \{>, \geq\}$ and the other $gop \in \{<\}$ or one $gop \in \{<, \leq\}$ and the other $gop \in \{>\}$).
2. The constants are different and both $gops \in \{=\}$.

Definition 4 can be seen as there being an assignment operation that falsifies the guard of the affected-by transition and Definition 5 can be seen as there being two guards that cannot be satisfied together. For example, consider subsequence t_5t_3 in the EFSM M (Fig. 1), t_5 requires v_3 to be less than zero whereas t_3 requires v_3 to be greater than zero. Also, t_5t_3 is a definition clear path for v_3 and therefore a TP that includes this subsequence is infeasible. This subsequence is infeasible regardless of the fact that t_5 cannot be executed in the first place since t_2 assigns to v_3 a positive value which opposes the guard of t_5.

By Definitions 3, 4 and 5, two cases can be defined where a TP is clearly infeasible:

Definition 6: A TP $t_1, t_2, .., t_n$ with length $n > 1$ is *definitely infeasible* if there exists $1 \leq i < j \leq n$ such that one of the following holds:

1. t_i is an affecting transition with operation $op \in op^{vc}$, t_j is an affected-by transition with guard $g \in g^{vc}$ and op opposes g.
2. the guards g_i and g_j of t_i and t_j respectively are of type g^{vc} and g_i opposes g_j.

3.1.1. Dependencies Representation and Penalties

This subsection describes the penalty values used in the feasibility metric. The aim is that a penalty value estimates how easily a given guard can be satisfied in the TP. Since a guard can be affected by a previous operation, we consider three factors when assigning a penalty to a pair of (affecting, affected-by). The first factor is related to the guard type. For example, a guard of type g^{vc} can be classified as the hardest since the option of selecting the values of either c or v is not available. In contrast, a guard of the type g^{pv} is typically easier to satisfy since it is possible to choose the value of the parameter. The second factor concerns the guard operator. For example, the operator = is normally the most difficult to satisfy and $\neq$ is the easiest. Finally, the third factor is related to the operation type of an affecting transition. For example, an operation of type op^{vp} is potentially useful since the parameter provides an opportunity to try to select a suitable value for v while op^{vc} is the worst since it is not possible to select the value of c. In addition to the penalty between a pair of (affecting, affected-by), it is possible to have a guard that is not affected by any operation (e.g. g^{pc}) and for such a case, only the first two factors are considered when assigning a penalty.

Table 1 shows the penalty values used in this work. For cases where there are no affecting transitions, '–' is used to indicate that the choices op^{vp}, op^{vv} and op^{vc} are irrelevant. Where a TP is definitely infeasible, INF[1] represents a large positive integer. The suggested penalty values are based on previous research [32] and initial experiments with one EFSM. Although the penalty values were found to be effective during the experiment, these values are by no means definitive and other suitable values can be used.

[1]INF represents a large positive integer to indicate that a given path is infeasible. In all experiments INF was set to be 1×10^4 since in the experiments the penalty values associated with transitions dependencies (see Table 1) cannot otherwise lead to a given TP being assigned a penalty value $\geq 10^4$. However, other large positive integers can also be used.

Table 1. The suggested penalty values where *INF* is a large positive integer to indicate that a given dependency represents an infeasible case.

Rows ID	Guard & operator	Assignment			
		(nop)	(op^{vp})	(op^{vv})	(op^{vc})
1.	$g^{pv}(=)$	4	8	16	24
2.	$g^{pv}(<, >)$	3	6	12	18
3.	$g^{pv}(\leq, \geq)$	2	4	8	12
4.	$g^{pv}(\neq)$	1	2	4	6
5.	$g^{vv}(=)$	16	20	40	60
6.	$g^{vv}(<, >)$	12	16	32	48
7.	$g^{vv}(\leq, \geq)$	8	12	24	36
8.	$g^{vv}(\neq)$	4	8	16	24
9.	$g^{vc}(=)$	40	30	60	INF if False and 0 otherwise
10.	$g^{vc}(<, >)$	32	24	48	INF if False and 0 otherwise
11.	$g^{vc}(\leq, \geq)$	24	18	36	INF if False and 0 otherwise
12.	$g^{vc}(\neq)$	16	12	24	INF if False and 0 otherwise
13.	$g^{pc}(=)$	12	-	-	-
14.	$g^{pc}(<, >)$	8	-	-	-
15.	$g^{pc}(\leq, \geq)$	4	-	-	-
16.	$g^{pc}(\neq)$	1	-	-	-
17.	$g^{pp}(=)$	6	-	-	-
18.	$g^{pp}(<, >)$	4	-	-	-
19.	$g^{pp}(\leq, \geq)$	2	-	-	-
20.	$g^{pp}(\neq)$	1	-	-	-
21.	g_i opposes g_j	INF	-	-	-

A guard can be given using nested IFs or predicates linked by AND and OR. For guards represented as nested IF or linked by AND we sum the penalties, however, the minimum penalty is used with an OR [35]. The dependency between affecting and affected-by transitions can occur on the basis of one or more context variables and an affected-by transition can be affected by one or more transitions in a TP. Therefore, each dependency between a pair of (affecting, affected-by) transitions is recorded together with the context variable at which the dependency occurs. There are three types of assignments and each type is represented by an integer. Integers -2 and -1 are used to mean an assignment of a constant (op^{vc}) and an assignment of a parameter (op^{vp}) respectively. However, a number in [1..*m*] represents the corresponding context variable appearing on the right-hand side of the assignment. If an assignment of type op^{vv} references more than one context variable, in the calculation we choose one of these: this reduces the time taken to compute the feasibility metric. We observe that if it is possible to easily set the value of one of the context variables then it may be less important whether it is possible to set the values of the others. Consider, for example, the problem of satisfying $v=v' + v''$ for context variables v, v' and v''. If the value of v' can easily be set using a parameter p, then it may be possible to choose a value for p that leads to the guard being satisfied. As a result of this observation, the referenced variable v_j is chosen by considering its assignment in the previous transition and using the following preference: (1) the assignment (to v_j) references a parameter, (2) the assignment references a constant and (3) the assignment references context variables. Having chosen the v_j, we use the penalty shown in Table 1. It is possible that there is no assignment (*nop)* and so no dependency between the transitions, or there is

Table 2. Assignment's types representation

op	Representation	Meaning
op^{vp}	-1	An assignment to v that references a parameter and no context variables
op^{vc}	-2	An assignment of a constant to v.
op^{vv}	$v_{1..\,m}$	An assignment to v that references context variables
nop	0	There is no assignment and so no dependency or open ended dependency

an open-ended dependency (a reference to a variable whose value is not defined). Such cases are represented by 0. Table 2 lists the dependency types and their representation.

Example 1: This example shows how the relationship between a pair of (affecting, affected-by) transitions is represented. The EFSM M (Fig. 1) has three context variables v_1, v_2, and v_3. Consider transitions t_2 and t_3: t_2 has an operation (op^{vc}) that assigns a constant value to v_3 whereas the guard of t_3 compares v_3 to a constant (g^{vc}) and so there is dataflow dependence between t_2 and t_3 at v_3. Further, there is no dependency on the other context variables v_1 and v_2. Therefore, t_2 is an affecting transition when it comes before t_3, which is an affected-by. From Table 1 (Row 10, Column 6), the associated penalty is zero. There is no dependency between t_2 and t_3 at v_1 and v_2. Thus, in this case we represent the dependency as a five-tuple. The first three fields record the dependency and penalty which occur at each context variable (in this example we have three context variables) and the fourth field, *gp*, records the sum of penalties of guards that do not involve context variables. The last field is a Boolean used to record whether there is a dependency between the considered transitions. The first three fields have two parts: the dependency type and the associated penalty value.

	Assignment type | Penalty		**t_2**	**gp: $g^{pc\&pp}$**	**Dependency?**
t_3	v_1= 0 | 0	v_2 = 0 | 0	v_3 = -2 | 0	0	True

The information in the above tuple can be read with the help of Tables 1 and 2 as: there is a dependency between t_2 and t_3 at v_3 where the dependency ends (when working backwards from t_3 to t_2) with an assignment of a constant value and the associated penalty is 0. In addition, there is no dependency at v_1 and v_2. Also, all guards of t_3 involve context variable and so the *gp* field has the value of 0.

The tuples are stored in a matrix, a *relation matrix*, to represent the dependencies and penalties among all the transitions in a given EFSM. The matrix has size n x n where n is the number of transitions in the considered EFSM. Affected-by transitions are rows whereas columns represent affecting transitions. Each cell in this matrix has the form described above and is computed once for the EFSM.

3.1.2. The Feasibility Metric

In this section we describe the feasibility metric. The aim is for low values to normally be associated with FTPs where it is relatively easy to find associated input sequences. The feasibility metric therefore penalises aspects that are likely to make it harder to find input to trigger a TP. The intention is to use this feasibility metric when searching for TPs that satisfy a test criterion: the search uses the feasibility metric to evaluate TPs.

Fig. 2 shows a description of the algorithm that calculates the feasibility metric. The inputs are the relation matrix and a TP with length $n > 1$. The algorithm first considers the

penalty of any guard that does not involve context variables (Line 10). It then treats the last transition as a potential affected-by transition and determines which previous transitions are affecting (Line 13). If the current pair of transitions (t_{n-1}, t_n) forms a pair of (affecting, affected-by) then a loop is entered (Line 16) to decide at which context variables there is a dependency or a penalty to be incurred. There are two cases: (1) The dependency type is in [-2..0], the related variable is set to be checked (Line 20) and if the corresponding penalty is greater than 0 (Line 21), this is added. (2) The dependency type is greater than 0 which means that the dependency may continue by an assignment referencing context variables, the related variable is set to be checked (Line 26), and if the corresponding penalty is greater than 0, then the dependency continues. Thus, the penalty is added and a call is made to *check* to detect all previous assignments that are propagated to the current context variable (Line 28).

A TP feasibility metric

```
1.  input: TP of length n, EFSM relation matrix
2.  output: non negative integer value
3.  goal: evaluate a TP complexity
4.  initialize: result := 0; bool array [1..v_k]
5.  begin
6.  for i := n downto first_transition
7.  begin
8.      bool array [1..v_k]:= false;
9.      j:= i;
10.     result := result + [t_i,t_j].gp;
11.    while (j > first_transition) do
12.    begin
13.       j := j -1;
14.       if [t_i,t_j].dependency == true then
15.       begin
16.          for v_s := v_1 to v_k do
17           begin
18.             if ([t_i,t_j].v_s(type) ≤ 0) and (not bool[v_s]) then
19.             begin
20.                bool[v_s] := true;
21.                if [t_i,t_j].v_s(penalty) > 0 then
22.      result := result + [t_i,t_j].v_s(penalty)
23.             end;
24.          if ([t_i,t_j].v_s(type) > 0) and (not bool[v_s]) then
25.          begin
26.             bool[v_s] := true;
27.             if [t_i,t_j].v_s(penalty) > 0 then
28.      result := result + [t_i,t_j].v_s(penalty) + check(t_i,t_j,v_s);
29.          end;
30.       end;
31.     end;
32.   end;
33. end;
34. return  result;
35. end.
```

Function *check* all of a transition dependencies

```
A1.  input: TP, t_i,t_j,v_s
A2.  output: non negative integer value
A3.  goal: trace back a flow dependence on variable v_s
A4.  initialise: result := 0; found := false;
A5.  begin
A6.     p := j + 1;
A7.     while (p > first_transtion) and (not found) do
A8.     begin
A9.        p := p – 1;
A10.       if [t_i,t_p].v_s(type) ≠ 0 then
A11.       begin
A12          case [t_i,t_p].v_s(type) of
A13.           -2   : result := result + 60;
A14.           -1   : result := result + 20;
A15.           1..k  : result := result + 40 + check(t_p, t_p-1, v_1..k)
A16.         end;
A17.         found := true;
A18.      end;
A19.   end;
A20.   if found then
A21.       return result
A22.   else return result + 60;
A23. end.
```

Figure 2. High level description of the algorithm that calculates the TP feasibility metric

The recursive *check* subroutine performs data dependence analysis by starting from the context variable and affecting transition passed to the call and then working backwards to find all previous transitions that may affect the value of the context variable (Line A10). If an earlier transition t_p is found to affect the context variable, then the subroutine finds the type of the assignment (Line A12). If the assignment type is found to be less than 0 then the context variable is assigned either a constant or a parameter value. Then the subroutine penalises referencing to a constant with 60 and to a parameter with 20 and stops (no earlier assignments affect this assignment). If the assignment type is greater than 0, the

assignment references a context variable v'. Here, the subroutine penalises this referencing by 40 and repeats the process by calling *check* with t_p and v' (Line A15). If the dependency is open ended (depends on an undefined initial value of a variable) then 60 is added (Line A22). When the subroutine stops (Line A21 or A22) it returns the sum of the penalties. After the current pair of transitions (t_{n-1}, t_n) is scanned, another cycle starts to detect any possible relation and penalty between the next pair (t_{n-2}, t_n) (Line 13) and so forth.

Example 2: Let's consider the feasibility metric calculation of the TP $t_1t_2t_3t_4$ through M (Fig. 1). For the considered TP, the following part of the relation matrix is required:

Pairs (*aff-by*, *aff*)	**Dependency at v_1**		**Dependency at v_2**		**Dependency at v_3**		**gp**	**Dependency ?**
	Type	**Penalty**	**Type**	**Penalty**	**Type**	**Penalty**		
t_1 affected-by t_1	-1	0	-1	0	0	0	16	False
t_2 affected-by t_1	-1	0	-1	0	0	0	0	False
t_3 affected-by t_1	-1	0	-1	0	0	0	0	False
t_4 affected-by t_1	-1	16	-1	16	0	0	0	True
t_3 affected-by t_2	0	0	0	0	-2	0	0	True
t_4 affected-by t_2	0	0	0	0	-2	0	0	False
t_4 affected-by t_3	2	32	0	0	2	0	0	True

The feasibility metric algorithm starts from t_4 and checks whether it has guards that do not involve variables, the *gp* field, see Fig. 1. However, t_4 does not have such guards, so the algorithm tests whether t_3 affects t_4. Since t_4 has a guard that references v_1, and t_3 has an assignment to v_1, there is a dependency between (t_3 (op^{vv}), t_4 (g^{vv})) at v_1. Since the dependency type is 2, the dependency continues through v_2. Here, the algorithm collects the penalty (32) (Row 6, Column 5 in Table 1) and calls *check*(t_4,t_3,v_1) to detect earlier transitions that affect the value of v_1 through v_2. The function *check* penalises this referencing by 40 and then computes *check*(t_3,t_2,v_2) to determine earlier transitions that affect v_2. From the relations matrix, t_2 does not affect the value of v_2, thus the function considers a possible earlier assignment and so it performs *check*(t_3,t_1,v_2). From the relation matrix, t_1 assigns a parameter value to v_2 (assignment type = -1). Thus *check* penalises this referencing by 20 and returns the total penalty to the main algorithm. The main algorithm continues by determining whether the pair (t_4, t_3) has dependencies on the remaining context variables v_2 and v_3. Since there are no such dependencies, the algorithm proceeds to the next pair (t_4, t_2) in the path. Since t_2 does not affect t_4, the next pair of transitions is checked (t_4, t_1). For this pair, there are two dependencies at v_1 and v_2 where both dependencies end by an assignment of a parameter value. However, a dependency at v_1 was previously detected, and so we know that the path from t_1 to t_4 is not definition clear for v_1. Thus only the dependency at v_2 is considered and the penalty (16) is collected (Row 6, Column 4 in Table 1). Since t_1 is the first transition, the algorithm has completed testing all the relations between t_4 and earlier transitions. Now, the algorithm proceeds to determine the dependencies between t_3 and the earlier transitions.

From the relation matrix, only t_2 affects t_3 at v_3, and the dependency ends by an assignment of a constant to v_3. The algorithm collects the penalty (0) (Row 10, Column 6 in Table 1) and continues with (t_3, t_1). Again, t_1 is the first transition and the algorithm has completed testing all the relations between t_3 and earlier transitions. Now, the algorithm proceeds to determine the dependencies between t_2 and the earlier transitions. Since t_2 does not have a guard, no penalty is incurred.

Finally, when the algorithm reaches t_1 to determine its relations with earlier transitions, it detects that t_1 has guards that involve only parameters and constants, thus the value of *gp* field (16) is added. Thus, the total penalty (124) of $t_1t_2t_3t_4$ is reported.

3.1.3. The GA Encoding for FTP Generation

The proposed FTP generation method uses the encoding technique from [36] in which a TP is represented by a sequence of integers, each number defining a transition. Given an EFSM with k states, let $n_1, n_2.. n_k$ be the number of transitions leaving each state. The method calculates the lowest common multiple *LCM* of $n_1, n_2,... n_k$. The last step is to define the ranges $r_1, r_2,... r_k$ for each state as $r_i = LCM / n_i$. An individual is a sequence of integers $i_1, i_2,...i_n$, each in [1.. *LCM*]. Each number i_j is divided by the corresponding r_j to determine the transition it defines. Using this encoding, every individual defines a TP.

An alternative is to use the transition label number to map a sequence of integers to a possible TP. However, this approach has the problem that not every sequence of integers defines a TP that is syntactically correct; it might contain consecutive integers i and j such that the transition represented by i cannot be followed by the transition represented by j. Therefore, there can be a large number of generated TPs that are redundant.

Example 3: The EFSM shown in Fig. 1 has $k = 3$ states, $n_1 = 2$, $n_2 = 2$ and $n_3 = 2$. Thus $LCM = 2$ and $r_1 = 1$, $r_2 = 1$ and $r_3 = 1$. If a sequence of integers is generated in the range [1..2] i.e. <2, 1, 2> then by starting from the first state, the first number represents t_2. Since t_2 ends at the second state, r_2 has to be used and so the second integer represents t_5. Similarly, t_5 ends at the second state and so by using r_2 the last number represents t_3. The TP is therefore $t_2t_5t_3$.

3.2. Phase 2: A Method for Generating an Input Sequence to Trigger an FTP

We have described a feasibility metric that aims to direct the choice of TPs towards those that are feasible and relatively easy to trigger. Given a test criterion we can search for a set of TPs, with good feasibility values, that satisfies the criterion. However, we still have to find input sequences to trigger these TPs and in this section we describe a search-based approach for solving this problem.

We need a fitness function to convert the problem of searching for a suitable test case into an optimisation problem. If a given path, within a program, consists merely of assignment statements any input can be used because the assignments form a single path from which the execution flow cannot divert. Problems arise when a program's path contains conditional statements such as (IF, FOR and WHILE) for which the execution flow may divert away from the test target. The work of Tracey et al. [37, 38] proposed a fitness calculation method in the presence of conditional statements which considered test case generation as a minimisation problem (see Table 3). This method is widely applied when generating test cases to satisfy a condition (predicate) in a program's path. Consider for example a predicate ($x < y$), for which the search should locate suitable values for both x and y. By referring to Table 3, the fitness value (also called a branch distance) is 0 when ($x - y < 0$) which states that the current values of x and y satisfy the given predicate. However, if the branch distance is not zero, it reflects how close the selected values were to achieving the predicate (branch distance = $x - y + k$; where $k > 0$ is a constant added

Table 3. Tracey et al. fitness calculations for different types of guards. The constant k, $k > 0$, is added when the guard is not satisfied.

Guard	Fitness Calculation
Boolean	if *TRUE* then 0 else k
$a = b$	if $abs(a - b) = 0$ then 0 else abs(a − b) + k
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else k
$a < b$	if $a - b < 0$ then 0 else $(a - b) + k$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + k$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + k$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + k$
$\neg a$	Negation is moved inwards and propagated over a

when the guard is not satisfied). Thus, the smaller the branch distance is the closer the selected values were to achieving the predicate.

Programs can have nested predicates and here the fitness function should reward a test case that achieves more predicates. As a result, using only the branch distance to guide the search can be insufficient and extra information is required. This is given in terms of approach level or approximation level proposed by Wegener et al. [34]. The approach of [34] is based on critical nodes, where a critical node is a conditional statement at which the execution flow may divert. Then, the approach level measures how close a test case was to executing the target statement by subtracting one from the number of critical nodes away from the target (Equation 2). Since achieving more predicates should result in a smaller (better) fitness, the branch distance is normalised to a value in the range [0..1] (Equation 1). In this way, the final fitness function consists of two components: branch distance and approach level as shown in Equation 3.

$$\textbf{norm}\ (branch\_distance) = 1 - 1.05^{-(branch\_distance)} \quad (\textbf{1})$$

$$\textbf{approach\_Level} = NumOfCriticalNodesAwayFromTarget - 1 \quad (\textbf{2})$$

$$\textbf{fitness} = approach\_level + norm\ (branch\_distance) \quad (\textbf{3})$$

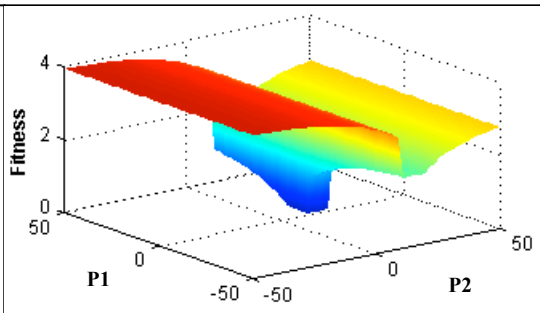
Consider, for example, t_1 in M (Fig. 1). This transition requires two inputs to satisfy four nested predicates. By applying the fitness calculation method proposed by [34] (Fig. 3a), the associated fitness function landscape (Fig. 3b) has a smooth downgraded surface. Such a landscape provides the search with adequate guidance to progress towards its goal.

The fitness calculation method proposed by [34] is effective in structural testing where the test target is represented as a single node in the main body of the function or the program. However, this technique is not designed to cope with the case when the test subject involves a sequence of calls to transitions. In this case, the test target is a sequence of sub-targets (each transition in an FTP) that have to be achieved. Since a transition in an EFSM can be considered to be a function with input parameters and conditions [33], the problem of generating an input sequence to trigger a given FTP can be seen as finding suitable input parameter values to be applied to each transition (function) in that FTP and in the order that each transition appears in this FTP.

In order to describe the proposed fitness calculation method, consider the problem of finding an input sequence that can exercise the FTP t_1t_1 through the EFSM M (Fig. 1). For

```
Double t1( int p1, int p2)
{if p1 >= 10
  if p1 <= 20
   if p2 >= 0
    if p2 <= 10
      result = 0 //Test Target achieved
    else result = Norm(abs(b - 10))
   else result = Norm(abs(b - 0)) + 1
  else result = Norm(abs(a - 20))+ 2
 else result = Norm(abs(a -10)) + 3 }
```

a. The fitness calculation



b. The associated fitness landscape

Figure 3: An example of a fitness calculation by using Wegener et al. (2001) approach.

such a path, the search should locate suitable input values (p_1,p_2) that successfully trigger the first transition, t_1, and then progress to find input values (p_3,p_4) that trigger the next t_1.

The manipulation of a path in this way is similar to the structure of nested IF statements where each IF statement compares the associated function's return value with 0. By applying the fitness calculation proposed by Wegener et al. [34] to each transition (function) in a path, if a function is successfully triggered then its return value is 0 otherwise the return value reflects the fitness of the input values in respect only to this

$$\textbf{function\_distance} = norm\ (branch\_distance) + approach\ level \quad (4)$$

$$\textbf{transition\_approach\_level} = NumOfCrticalTransAwayFromTarget - 1 \quad (5)$$

$$\textbf{path\_fitness} = norm\ (function\_distance) + transition\_approach\_level \quad (6)$$

particular function. Let's refer to the return value of a function by *function_distance*. The first transition in the path can be considered as the upper IF statement and then functions which come next are treated as nested IF statements. Therefore, the fitness function for a path can be derived in a similar way to the method proposed by Wegener et al. [34] for nested predicates. That is, given an FTP, the function distance is calculated for each guarded transition by applying the Wegener et al. approach (Equation 4). Then, any transition that has guard(s) is considered a critical transition and so the value of *function_approach_level* is derived by subtracting 1 from the number of critical transitions away from the target transition (Equation 5). Finally, the path fitness is the sum of *function_approach_level* and the normalised value of *function_distance* at the transition where the execution flow was diverted (Equation 6).

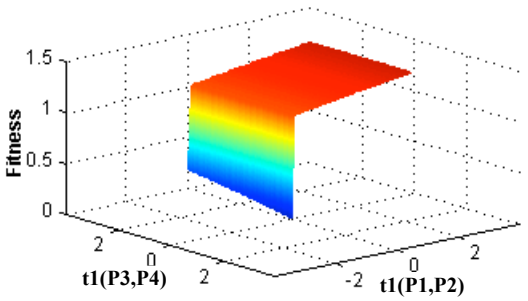
Let's consider the path $t_1(p_1,p_2)$ $t_1(p_3,p_4)$ in M (Fig. 1). By applying the proposed fitness calculation (Fig. 4a), the associated fitness function landscape (Fig. 4b) appears to have a smooth and downgraded surface, which can provide search with guidance towards its goal.

In an EFSM, a transition's guards can be sequenced as nested IF statements (as shown in Figure 1.4) or linked by AND and OR. In order to apply the proposed feasibility metric, guards linked by AND operators are represented as nested IF statements.

If guards are linked by OR, a transition is split into a number of transitions equal to the number of OR operators + 1. One benefit of this is that the test considers satisfying each predicate/condition in a guard. However, the alternative would be to use the

```
double Path_t1_t1(int p1, int p2, int p3,
int p4)
{if t1 (p1, p2) = 0
  if t1 (p3, p4) = 0
    result = 0 //Target achieved
  else result = Norm(t1(p1,p2))
 else result = Norm(t1(p3,p4)) + 1
}
```

a. The proposed fitness calculation



b. The associated fitness landscape

Figure 4: An example of calculating a path fitness approach by using the proposed approach

minimum fitness value for a set of conditions linked by OR operator as proposed by [35].

3.2.1. GA Encoding for Input Sequence Generation

A solution encoding can be selected on the basis of the input parameter types. It is possible to use *binary* or *integer* encoding when all of the considered input parameters are *integers*; however, if some of the input parameters are of *double* data type then real valued encoding can be used. A candidate solution that represents a test sequence consists of components where each component represents one input parameter. For example, a possible solution encoding of the path shown in Fig. 4 consists of four components of type integer.

4. Experiment

In this section we describe experiments in which the proposed technique was used to generate test cases from five EFSMs and the results compared with a random approach.

4.1. Experimental Design

The experiments used five EFSMs: Lift EFSM, In-Flight safety EFSM, ATM EFSM, Class 2 protocol EFSM and Inres initiator EFSM. Both Lift and In-Flight EFSMs are synthesised case studies whereas the rest of the EFSMs were taken from the literature. Details of these EFSMs are given in Appendix A.

In designing the experiment, we aimed to evaluate the effectiveness of the proposed approach. In order to achieve this, we considered two factors, the first relating to the length of the TPs. That is a short TP is likely to be easy to trigger since it has few guards and operations. Since the subject EFSMs had 15-31 transitions, we considered TPs of lengths 9, 12 and 15 to be sufficient to avoid the impact of this factor. We used these values since for each transition t there was a TP of length 9 or less that contained t. However, the approach is designed to generate any possible TP length and so can be used to produce other TP lengths. The second factor is related to the EFSMs used. We used a random approach to give an indication of the difficulty of test case generation for these EFSMs. For each EFSM we use a random approach with two phases that are similar to the proposed approach. The first phase generates random TPs from each EFSM and then the second phase generates random input sequences to attempt to trigger the randomly generated TPs.

For each EFSM, we generated three sets of TPs using the proposed approach. The first set contained TPs of length 9, the second contained TPs of length 12 and the third contained TPs of length 15. Each set provided a transition coverage test suite for the

considered EFSM. That is, each TP in a set is responsible for covering one transition in the EFSM. Naturally, this leads to more TPs than is necessary but the redundancy provides additional experimental subjects. For ease of reference, each of the three sets of TPs that was derived from an EFSM using the proposed approach will be referred to as a *group* of TPs. After TPs were generated, the second phase of the proposed approach was applied to each generated TP to try to generate an input sequence that can execute this TP.

Similarly, the random approach was also applied to each EFSM to generate three similar sets of TPs (*alternative group* of TPs); for each *group* of TPs there was an *alternative group* of TPs containing TPs with the same length and each set provided transition coverage. Then, the second phase of the random approach was applied to each randomly generated TP to try to execute it.

When trying to execute the generated TPs, if a given TP was executed, then this is an FTP. If a given TP was generated and had a TP feasibility metric > INF, then this TP is definitely infeasible. If a generated TP had a TP feasibility metric < INF and a test case could not be generated to execute this TP (all attempts to trigger this TP were unsuccessful) then we manually inspected this TP to decide whether it was infeasible.

The proposed search-based approach and the random approach were implemented by using the publicly available Genetic and Evolutionary Algorithm Toolbox for Matlab GEATbx [39]. A detailed description of each of the GEATbx parameters used is beyond the scope of this paper. However, these parameters are fully explained at the GEATbx website [39] and we record the values used here to allow the experiment to be replicated.

For the proposed approach, an integer valued encoding was used to represent individuals. The population size was 100, a value determined through preliminary experiments. However, by using the approximation in [66] of Goldberg's work [65] for real valued genetic algorithms we get a population size of O(100) for a ten gene problem where each gene consists of 10 bits (1024 possible values), which matches our choice almost exactly. Selection was linear-ranking with selective pressure set to 1.8. Discrete recombination was used to recombine individuals. Discrete recombination between two individuals (parents) x: $x_1..x_n$ and y: $y_1..y_n$ is performed by selecting one component from one of the parents at a time (either x_i or y_i) with equal probability. The mutate integer method was used for mutation where each component from a given individual is mutated with a probability equal to 1/ number of components in this individual. This value for mutation means that each child has a relatively small amount of change done to it by mutation (on average one change), and thus will not undo the work by the crossover operator. For TP generation, each individual consisted of 9, 12 or 15 integers for the three sets of TPs. The range of values allowed for each variable varied according to the EFSM as previously described in Subsection 3.1.3. For input sequence generation, each individual consists of 25 integers, which represent the maximum number of input parameters required by a TP. The range of values allowed for each variable was [0..1000]. Thus, the input domain used with each TP had 1×10^{75} possible candidate solutions. Search terminated if the fitness value of zero was achieved or a maximum number of 1000 generations was reached. Finally, for each TP we repeated the search ten times.

For the random approach, GEATbx allows the use of a standard random approach by setting the recombination and mutation methods to "recnone" and "mutrandint"

respectively. The rest of the settings in terms of individuals encoding, population size and maximum number of allowed generations were the same as that of the proposed approach.

4.2. Experimental Results

Fig. 5 shows the results for the five EFSMs. A total of 72 TPs were generated for the Lift EFSM. Fig. 5a shows that all of the TPs were FTPs. The average TP feasibility metric value of the generated FTPs was approximately 127. Furthermore, Fig. 5a shows that the GA search that generated input sequences always required more than 600 generations[2].

For the In-Flight EFSM, 93 TPs were generated. Fig. 5b shows that all of these TPs were FTPs with an average TP feasibility metric value of 113. In the GA search that generated the input sequences, some FTPs were triggered as early as one generation. This suggests that the first phase generated TPs that are easy to trigger in the second phase.

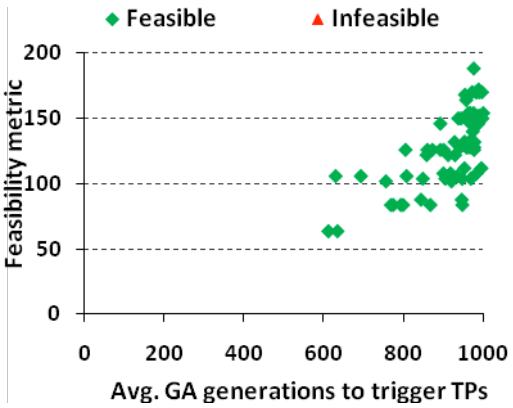
The results for the ATM EFSM are shown in Fig. 5c. 87 of the 90 TPs were FTPs and 3 were infeasible. The FTPs had an average feasibility metric of 55. The GA search for input sequences did not require more than 600 generations and many FTPs were triggered relatively quickly. The three infeasible TPs had a feasibility metric value of 208; they were not identified as definitely infeasible. When we examined the ATM's transitions, we found that the guard of t_3 is (*attempt* =3) and this cannot be satisfied unless t_2 previously occurred three times. Such behaviour cannot be easily estimated (by penalty values) and so any TP that included t_3 is likely to be infeasible. All of the infeasible TPs included t_3 without sufficient occurrences of t_2.

For the Class 2 EFSM, there were 63 generated TPs and all these TPs were FTPs as observed in Fig. 5d. The average TP feasibility metric for these FTPs was 15. Compared to the previous EFSMs, the Class 2 EFSM appears to have better TP feasibility metric values. Fig. 5d shows that the maximum number of generations did not exceed 30.

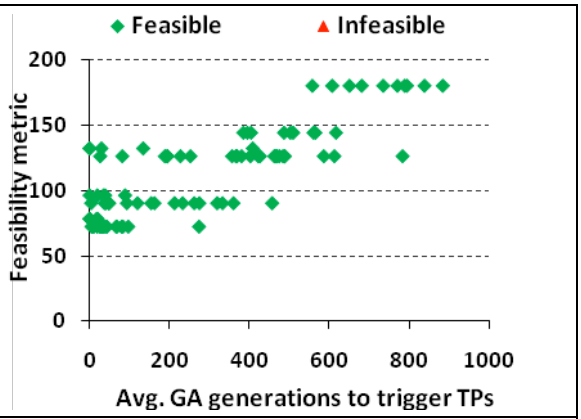
For the Inres initiator, 36 of the 45 TPs were FTPs. The average TP feasibility metric value of the FTPs was 7. Similar to the previous EFSM, for all FTPs, the GA search that generated input sequences did not require more than 35 generations. However, 9 TPs were infeasible but associated with a TP feasibility metric value < INF. We found that transitions t_4, t_9 and t_{12} have guard (*counter* $\geq$ 4) that references a counter variable. Any TP that includes one of these transitions requires other transitions to previously occur a certain number of times. For example, t_3 must occur exactly four times before the t_4 can be taken. Similar to the ATM, TPs that included one of t_4, t_9 or t_{12} were infeasible.

Table 4 summarises the results derived from the five EFSMs. From this table, the Lift EFSM is associated with the highest average FTPs feasibility metric value whereas the Inres initiator EFSM has the lowest average FTPs feasibility metric value. By calculating the average number of generations that was required to trigger all the FTPs in each EFSM, Fig. 5f plots the average feasibility metric of each EFSM as reported in Table 4 against the average number of generations required to trigger all the FTPs in that EFSM.

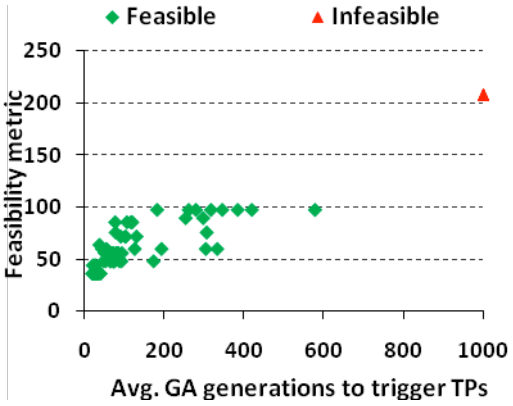
[2]All experiments were conducted on a PC with Windows® XP Service Pack 3 OS, Intel® Pentium® 4 CPU 2.80 GHz 2.79 GHz and 1.24 GB of RAM



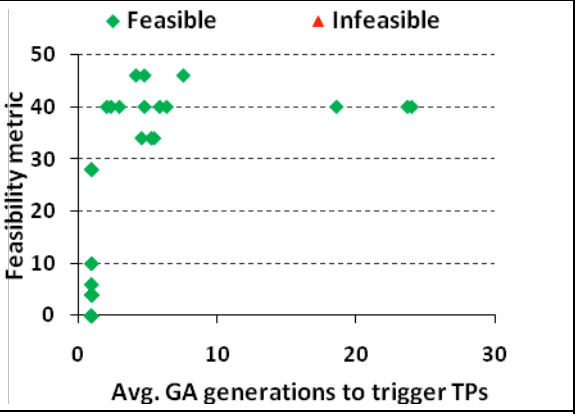
a. The group of TPs derived from the Lift EFSM



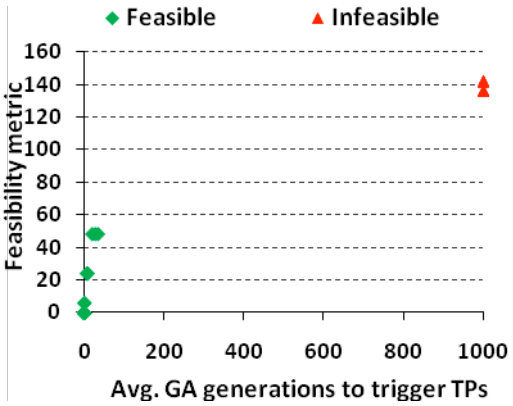
b. The group of TPs derived from the In-Flight EFSM



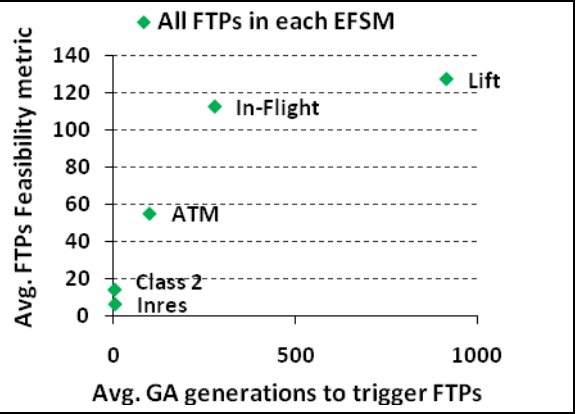
c. The group of TPs derived from the ATM EFSM



d. The group of TPs derived from the Class 2 EFSM



e. The group of TPs derived from the Inres Initiator EFSM



f. Each dot represents the average feasibility metric of all FTPs in one EFSM vs. average GA generations to trigger all FTPs in the same EFSM

Figure 5: Results of the proposed approach on the five EFSM case studies. Graphs *a*, *b*, *c*, *d* and *e* plots TPs feasibility metric vs. the average generations of the GA search to generate an input sequence to trigger the FTP in ten tries. Plot *f* shows (for each EFSM) the average feasibility metric of all FTPs vs. the average GA generations to trigger all the FTPs.

Table 4: Summary of the results achieved by the proposed approach on five EFSM case studies

EFSMs	Total TPs	FTPs	Infeasible TPs	Avg .FTPs Feasibility	Success rate	
					FTPs generation	Input sequence generation
Lift	72	72	0	≈ 127	= 100%	= 100%
In-Flight	93	93	0	≈ 113	= 100%	= 100%
ATM	90	87	3	≈ 55	≈ 96.7%	= 100%
Class 2	63	63	0	≈ 15	= 100%	= 100%
Inres	45	36	9	≈ 7	= 80%	= 100%
Total	363	351	12	–	≈ 96.6%	= 100%

From Fig. 5f, there is clearly a trend between the average feasibility metric of an EFSM and the average number of generations required to trigger the FTPs in this EFSM.

Table 4 also shows that for the Lift, In-Flight and Class 2 EFSMs, the proposed approach had a success rate of 100%. However, for the ATM and Inres initiator EFSMs, the success rate of the first phase of the proposed approach was 96.6% and 80% respectively and the success rate of the second phase was 100% for both EFSMs. By considering all the EFSMs, the total success rate of the first phase was 96.7% and for the second phase was 100%.

The first phase of the proposed approach on ATM and Inres initiator did not achieve 100% success rate due to both EFSM suffering from the counter problem. The counter problem seems to impact the TP generation phase (the proposed TP feasibility metric) since a TP that references a counter variable requires a specific transition sequence in order to be feasible. It appears to be difficult to identify such cases using static data flow dependencies formulated as penalty values. The counter problem requires additional analysis and an initial approach to overcome the counter problem was introduced in [40].

Table 5 summarises the results achieved with the random approach. For both Lift and In-Flight EFSMs, the random path generator could not generate any feasible TPs and so the random approach had a success rate of 0%. For the ATM EFSM, the random approach generated 49 FTPs out of 90 TPs (success rate ≈ 54.4%). However, the random input generator failed to trigger any of these 49 FTPs (success rate = 0%). The random path generator performed similarly on the Class 2 EFSM where it generated 37 FTPs out of 63 TPs (success rate ≈ 58.7%). Furthermore, the random input sequence generator was able to trigger 32 FTPs (success rate ≈ 86.5%). For the Inres initiator EFSM, the random approach generated 7 FTPs out of 45 TPs (success rate ≈ 15.5%) and triggered 5 FTPs (success rate ≈ 71.4%). Finally, the overall success rate associated with random path generator was approximately 26.6% and that associated with random input sequence generator was approximately 39.8%.

Table 5: Summary of the results achieved by the random approach on five EFSM case studies

EFSMs	Total TPs	FTPs	Infeasible TPs	Avg .FTPs Feasibility	Success rate	
					FTPs generation	Input sequence generation
Lift	72	0	72	–	= 0%	= 0%
In-Flight	93	0	93	–	= 0%	= 0%
ATM	90	49	41	≈ 219	≈ 54.4%	= 0%
Class 2	63	37	26	≈ 41	≈ 58.7%	≈ 86.5%
Inres	45	7	38	≈ 19	≈ 15.5%	≈ 71.4%
Total	363	93	270	–	≈ 25.6%	≈ 39.8%

The results achieved with the random approach show that generating FTPs and input test sequences from the considered EFSMs is not an easy task. Nevertheless, the results achieved by the proposed approach showed a significant success rate and demonstrated the effectiveness of the proposed search-based approach.

5. Statistical Study

This section describes the results of experiments that explored the relationship between the feasibility metric value of a TP and the effort required to find test input to trigger this TP.

5.1 Design

The results of the proposed approach, discussed in Section 4, seem to indicate that the proposed search-based approach generates FTPs whose feasibility metric value reflects the effort (such as time or number of generations) required to find input sequences to trigger them. Furthermore, it is useful to understand whether it is possible to predict the effort required to find an input sequence to trigger a TP by considering only its feasibility metric value. Thus the aim of the statistical study reported in this section was to answer two questions: For the FTPs generated by the proposed approach (1) is there a correlation between FTPs feasibility metric values and the effort (time and / or generations) required to trigger this FTP? (2) Can an FTP feasibility metric value be used to predict this effort? If the answers to these questions are positive then there is scope to use the feasibility metric to direct the choice of TP towards those for which it is easier (or harder) to find test input. In order to answer these two questions, there are two factors to be considered.

The first factor is related to using alternative input sequence generators. In the experiment, we used a GA to generate input sequences to trigger the generated TPs. It is useful to use another (non search-based) approach to generate such input sequences and we used a constraint solver. The second factor is related to the results observed from the experiment. We performed two sets of analysis, one where we analysed the results as a whole (non-clustered in Section 5.2) and the second where we grouped and averaged results with the same feasibility metric value (clustered in Section 5.3). The motivation behind this was to see if there was any bias in results with the same fitness, i.e. if a particular fitness value appeared more than once in the results data, then there would be more data rows corresponding to that row. Grouping the data according to feasibility metric value ensures that each unique feasibility metric value only appears once. Averaging also tries to address any high variability in the results and any over large outliers.

5.1.1 Using Constraint-Based Testing (CBT) to Generate Input Sequences

Constraint-Based Testing (CBT) expresses the problem of test case generation in terms of solving a set of constraints. These constraints are derived by symbolically executing a given path [41]. If a path is symbolically executed, the resultant constraints can be of two types: equality constraints and inequality constraints. Let e and e' be expressions:

1. An equality constraint can be given as $e = e'$ where both e and e' are constants, or e contains input parameters and e' is a constant.
2. An inequality constraint can be given as $e \leq 0$ where either e is a constant or e contains input parameters.

Given a set of equality and inequality constraints, a solver can be applied to try to find values of the parameters for which all the constraints hold. As mentioned previously, an FTP can be seen as functions to be called in a sequence. Given this description, a set of constraints from a given path can be derived through the following steps:

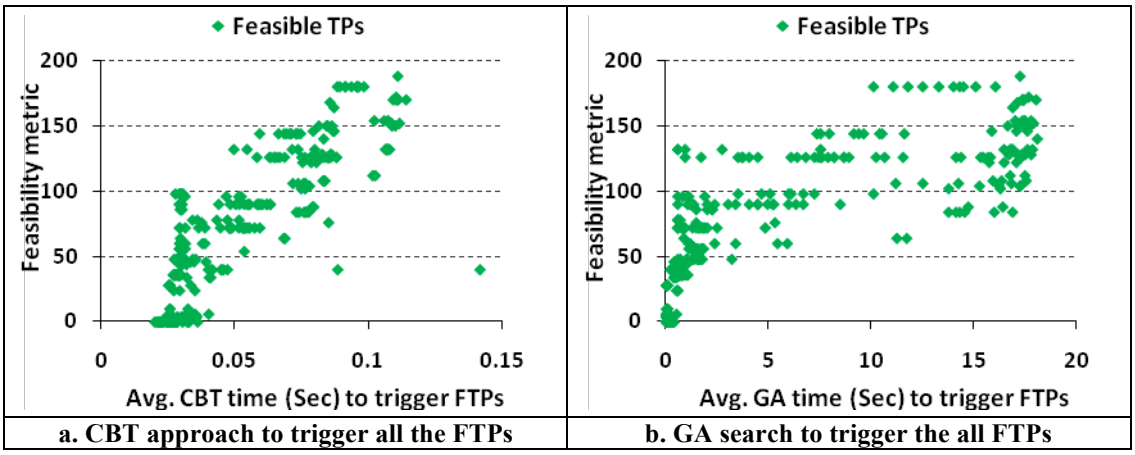
1. Rename the input parameters for each transition in the path so that all the input parameters have unique names.
2. By starting from the first transition, for each transition, then for each assignment statement, replace a context variable by the expression that is assigned to it using the input parameters and current values of context variables. If the transition contains guards, then for each guard that involves a variable, replace this variable by its current value in terms of parameters and constants.
3. If there is a transition that still has guards that reference context variables, the given path cannot be executed since the values of these variables are not yet defined. Otherwise, the resultant list of guards is a set of constraints that reference only input parameters and constants.

The set of constraints can be given to a solver in order to try to find suitable values for the input parameters included in the constraints. If the set of constraints are solved, the values returned by the solver comprise an input sequence that can exercise the considered FTP.

For convenience, we will refer to the proposed input sequence generator by GA and to the alternative constraint-based testing one by CBT. For the CBT approach, each FTP was transformed to a set of constraints and a solver, constrained nonlinear minimisation *fmincon* [42], was applied. A detailed description of the solver's parameters is provided in the MATLAB website; the values that were used are recoded here to allow replication. The range of values that the solver can search was set to [0..1000] while the initial solutions of the considered variables were randomly generated in the range [0..1000]. For each FTP, the solver was called ten times to try to find an adequate input sequence and then the average time required by the solver in the ten tries was calculated.

In order to answer the two questions, statistical software was used. For the first question, Pearson correlation was computed between FTP feasibility metric values and the CBT time, the GA time, and the GA generations required to trigger these FTPs. In Pearson correlations, there are two main outputs. The first output is the correlation coefficient r and the second output is the p value. The r value can be in the range [-1..1] where the range boundaries state a perfect correlation. Other values of r are classified to three categories [43]: (1) a small correlation when $0.10 \leq r \leq 0.29$, (2) a medium correlation when $0.30 \leq r \leq 0.49$ and (3) a large correlation when $r \geq 0.50$. The p value determines the confidence in the results where $p < 0.05$ denotes a statistically significant result.

We performed a linear regression. In linear regression, there is a dependent variable and one or more independent (or explanatory) variables. In this case we had only one independent variable, the feasibility metric value, and we used either GA time or CBT time as the dependent variable. Also, there are two hypotheses: the null hypothesis which states that the independent variable has a zero impact on the dependant variable whereas the other hypothesis states that the independent variable does impact the dependent one.



a. CBT approach to trigger all the FTPs | **b. GA search to trigger the all FTPs**

Figure 6. The performances of the GA and CBT approaches on all subject FTPs.

The important outputs of a linear regression are:

1. The coefficient of determination or *R Squared* (*RS*) which states how much the independent variable is capable of explaining the variance in the dependent variable. The range of *RS* values is [0..100]%.
2. The *F* ratio significance (*Sig*) which states the confidence (1 − *Sig*) by which the null hypothesis can be rejected (usually when *Sig* < 0.05).
3. The line fit plot which shows the trend by which the estimation can be performed.

5.2. Statistical Study on FTPs without Clustering

5.2.1. Correlation Study

The CBT approach described in the previous subsection was applied to each generated TP ten times in order to try to solve the TP constraints. The generated TPs for each EFSM are the same TPs generated from the first phase and reported in Fig. 5. The results showed that the CBT approach was successful in triggering all the FTPs that were generated from the five EFSM case studies. Infeasible TPs that were identified previously were also reported as unsolvable constraints by the solver. Fig. 6a shows the performance of the CBT approach on all the FTPs in terms of the average time that was required by the solver to trigger each FTP. Fig. 6b shows the performance (observed previously on Fig. 5) of the GA search that generated input sequences to trigger the same FTPs. From both Fig. 6a and Fig. 6b, it is clear that the CBT approach was much faster than the GA search. This could be explained by noticing that all the guards in the considered EFSM are linear, which assists CBT. Furthermore, guards that includes the equal operator {=} are known to be the hardest for a GA search to satisfy. However, for the CBT technique they often are simply assignments that must be applied before trying to solve the remainder inequality constraints. Although the CBT approach was found to be more efficient than the GA search, the applicability of the CBT approach remains the main concern. For example, solving a set of non-linear constraints over integer variables is generally undecidable [44]. There are at least two additional factors that may have made the CBT approach faster. First, we separately symbolically executed the TPs: the results might have been quite different if we had used a tool for symbolic evaluation and included, in the CBT results, the time it took. Second, the CBT approach was implemented using a professional CBT

Table 6: FTPs (no clustering) - Correlation among FTPs' feasibility metric, GA average generations, GA average time and CBT average time.

	Feasibility metric	GA Gen.	GA time	CBT time
Feasibility metric	1			
GA Gen.	0.791*	1		
GA time	0.798*	0.999*	1	
CBT time	0.864*	0.847*	0.851*	1

*** Correlation is significant at the $p < 0.01$ level (2-tailed).**

tool while the GA was applied using our prototype tool. As a result of these two factors, it is not surprising that CBT was significantly faster than the GA and the trends are of most interest. However, it may transpire that a mixed approach is sometimes best, in which paths are chosen using the GA and test input generated by CBT.

Table 6 shows the correlations from the FTPs in the GA group. Here, all the correlation values are statistically significant at $p < 0.01$ (shown by a *). The achieved correlation shows that there was a strong positive correlation ($r = 0.798$) between the feasibility metric and the required GA time in seconds to trigger the FTPs. Similarly, there was a strong positive correlation ($r = 0.791$) between the feasibility metric and the required GA generations to trigger the FTPs. Unsurprisingly, the relationship between the GA time and GA generations was found to be almost perfect ($r = 0.999$). Furthermore, the correlations between the feasibility metric and the required CBT time in seconds to trigger the FTPs was also found to be positive and strong ($r = 0.864$). This shows that for the considered test cases generators (GA and CBT), there was a strong agreement on their performances ($r = 0.851$). Greater feasibility metric values were associated with more GA time or CBT time. It was interesting to see the strong correlation between CBT time and GA time/generations.

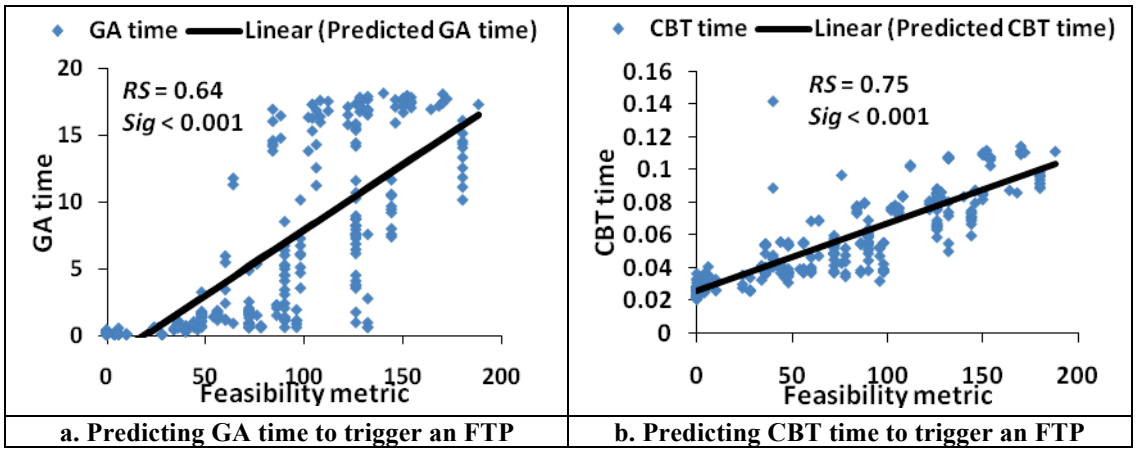
5.2.2. Regression Analysis

Liner regression analysis was performed on the generated FTPs. The independent variable was the feasibility metric whereas the dependent variables were the GA time and CBT time. The number of GA generations was not considered in this analysis since this was found to have an almost perfect correlation with GA time.

Fig. 7 reports the results of the regression analysis in terms of *RS* and *Sig* values and the fitness line fit plot. Fig. 7a and Fig. 7b show the predictions of GA time and CBT time respectively. From Fig. 7a, the regression analysis reported that the feasibility metric contributes significantly to the prediction of the GA time ($Sig < 0.001$) and the null hypothesis is rejected. Furthermore, the feasibility metric explains 64% of the variance in the GA time. Fig. 7b also shows that the feasibility metric contributes significantly to the prediction of the CBT time ($Sig < 0.001$). The feasibility metric here explains 75% of the variance in the CBT time.

5.2.3. Summary of the Statistical Analysis on FTPs without Clustering

The results of the correlation study can answer the first question about the relationship between the feasibility metric and how much effort, in terms of time, that is required by an input sequence generator to perform. There was a strong positive correlation between the feasibility metric and the time required by an input sequence generator (CBT or GA) to trigger the FTPs. The strength of the relationship was almost the same even though



a. Predicting GA time to trigger an FTP | b. Predicting CBT time to trigger an FTP

Figure 7. The line fit plots for predicting GA time and CBT time to trigger an FTP.

different input sequence generators were used. Greater FTP feasibility metric values were found to be associated with more time by an input sequence generator to trigger an FTP.

The results from the regression analysis led to an important finding that answers the second question. The feasibility metric has the potential to predict the effort of an input sequence generation approach (explains from 64% to 75% of the variance in effort).

5.3. Statistical Study on Clustered FTPs

In this study, the TPs with the same fitness were clustered. For each cluster, the corresponding GA time, GA generations and CBT time values were averaged. The correlation study was then conducted again on the clustered FTPs.

5.3.1. Correlation Study

Table 7 shows the correlation results obtained from the clustered FTPs. All the achieved correlations were statistically significant at $p < 0.01$. The feasibility metric had strong positive correlations with both the GA time ($r = 0.851$) and GA generations ($r = 0.847$) respectively. Furthermore, the correlation between the feasibility metric and the CBT time was also positive and strong ($r = 0.904$). Moreover, more GA time or generations was strongly associated with more CBT time ($r = 0.919$).

5.3.2. Regression Analysis

The linear regression was applied to the clustered FTPs. Fig. 8 reports the linear regression analysis of the feasibility metric and the GA time and also of the feasibility metric and the CBT time. Fig. 8a shows that the feasibility metric contributes significantly to the prediction of GA time ($sig < 0.001$). Furthermore, the feasibility metric is found to explain 72% of the variance in the GA time. Similarly, Fig. 8b reports that the feasibility metric makes a significant contribution to the prediction of the CBT time ($sig < 0.001$). 81% of the variance in the CBT time is explained by the feasibility metric.

5.3.3. Summary of the Statistical Analysis on Clustered FTPs

The results of the correlation study on the clustered FTPs answers the first question in a similar way to that derived from FTPs without clustering. There was a strong positive correlation between the feasibility metric and the time required by an input sequence generator (CBT or GA) to trigger the FTPs. Compared to the correlation results obtained

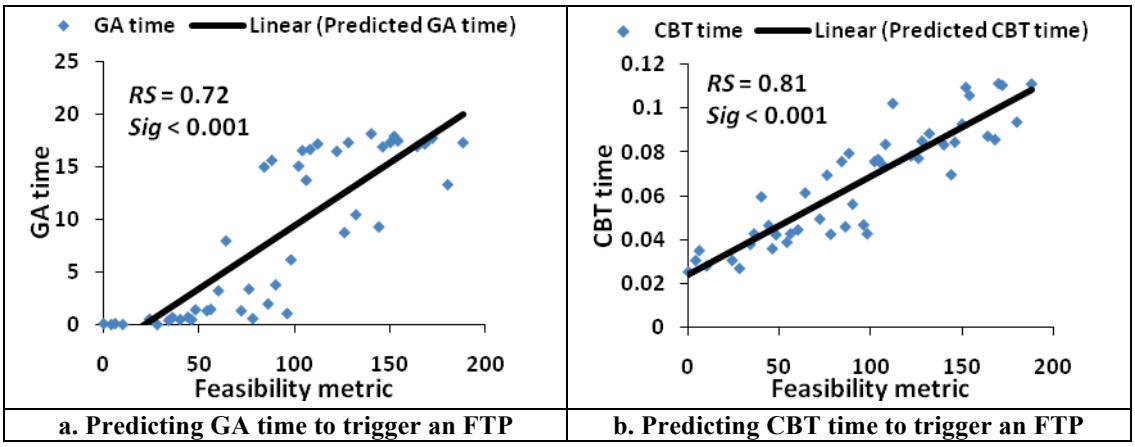
Table 7: FTPs (clustered) - Correlation among FTPs' feasibility metric, GA average generations, GA average time and CBT average time.

	Feasibility metric	GA Gen.	GA time	CBT time
Feasibility metric	1			
GA Gen.	0.847*	1		
GA time	0.851*	1.000*	1	
CBT time	0.904*	0.919*	0.919*	1

*** Correlation is significant at the $p < 0.01$ level (2-tailed).**

from the FTPs without clustering, these values clearly show an improvement in the strength of these correlations (see Table 6). Therefore, clustering the FTPs has led to stronger associations among the considered factors.

For the second question, the regression analysis on the clustered FTPs also provides a similar answer to that observed on the FTPs without clustering. The feasibility metric has the potential to predict the effort of an input sequence generation approach (explains from 72% to 81% of the variance in efforts). Compared to the linear regression analysis that was performed without FTPs being clustered (see Fig. 7), the feasibility metric is better able to predict both the GA time and CBT time when clustering was applied.



a. Predicting GA time to trigger an FTP | **b. Predicting CBT time to trigger an FTP**

Figure 8. The line fit plots for predicting GA time and CBT time to trigger an FTP.

5.4. Threats to validity

In this subsection we discuss the potential threats to the validity of our study. Threats to external validity are the conditions that restrict our ability to generalise our results. The main threats to external validity of this study are related to (1) the proposed TP feasibility metric penalties and (2) the achieved experimental results.

The proposed TP feasibility metric penalties, though they led to significant outcomes, are by no means definitive and research would be useful to calibrate these values further. It would also be interesting to run sensitivity analysis simulations to determine how robust the method is to small random changes in the penalties. In addition, there may scope to adapt them to properties of the EFSM and, in particular, its size. It is particularly important to note that INF, used in the feasibility metric, was given a value that ensured that any TP determined to be definitely infeasible would have a higher (worse) fitness value than one that was not. If longer paths are to be used then the value of INF should be updated in order to ensure that this property still holds. In addition to the

choice of penalty values, we only used one metaheuristic search technique. It would therefore be interesting to explore how effective the approach is when using other metaheuristics such as Simulated Annealing or Hill-Climbing.

Since the experimental results were derived from five EFSM case studies, we do not know to what extent the performance can be generalised. Three of the EFSMs were taken from the literature while we produced the other two. We used EFSMs for two different classes of system: two protocols and three control systems. The use of EFSMs for different types of system provides us with some additional confidence that the results will generalise. However, there would be value in repeating the experiments with additional EFSMs, although many studies use fewer than five EFSMs due to the shortage of published EFSMs that have non-trivial guards.

An additional factor that could influence the experimental results is the parameters used in the GA. We tried to limit this effect by basing the values used on recommendations in the literature. However, these values might not work well for other case studies. It would be interesting to perform a sensitivity analysis to determine how the results of the experiments change as we vary the GA parameters.

The main internal threat relates to the possibility of faults in our prototype tool, the CBT tool, or the tool used for the statistical analysis. We reduced the scope for faults by using commercial tools to provide the GA, the CBT, and the statistical analysis. In addition, we carefully checked and tested our implementation.

6. Related work

Many test generation approaches for systems modeled as EFSMs appear in the literature [16] , [17] , [18] , [36], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56], [57]. A major portion of the previous work can be categorised to three main groups:

1. Rewriting an EFSM to construct another form of EFSM which does not suffer from the path feasibility problem, see e.g., [16, 17, 45, 46].
2. Converting an EFSM to an FSM so that FSM-based testing techniques can be applied, see e.g., [45, 47-49].
3. Using symbolic execution and constraint satisfaction methods to check path feasibility and to generate path test data, see e.g., [50-52, 54].

The first category aims to overcome the problem by producing an EFSM in which all paths are feasible. However, there is no general algorithm for solving this problem. The automated approaches impose significant restrictions on the EFSMs. For example, they require that all actions and guards are linear. Thus, while approaches such as those described in [16, 17], are extremely useful for some classes of system, they lack generality. In addition, the work in this area did not consider the problem of generating test cases to cover the FTPs produced from the transformed EFSM. Finally, it may be difficult to use the transformed EFSM, in which all paths are feasible, as the basis of producing test cases that satisfy a test criterion expressed in terms of the original EFSM.

There are two main approaches within the second category: either the data is expanded to form an FSM or the data is abstracted out. The first approach can lead to the state explosion problem: the size of the resultant state space is exponential in the number of variables. The second approach can suffer from the path feasibility problem since a path in the FSM formed by abstracting an EFSM may not correspond to an FTP in the original EFSM [58]. The approaches that abstract the EFSM to produce an FSM are part of the

motivation for the work in this paper: we may be able to adapt these approaches to use the proposed feasibility metric. This would lead to new test generation methods that produce paths from the abstracted FSM that are likely to be feasible in the corresponding EFSM. Naturally, a method might be iterative: if a chosen path is found to be infeasible, or we fail to find a test case to trigger it, then we might produce a different path with good fitness.

Finally, approaches based on the last category are affected by the applicability limitations of symbolic execution and constraint satisfaction techniques [4, 44]. For example, solving a set of non-linear constraints over integer variables is generally an undecidable problem [44].

There are also other approaches that apply program testing techniques to test from an EFSM [18, 55]. An approach which employs software dataflow testing to derive a test sequence from EFSM models is presented in [18]. The selection of each test case depends on identifying all the associations between each output and all the inputs that affects that output. However, the approach essentially defines a test criterion rather than showing how test cases can be produced to satisfy a test criterion. Work has also generated test sequences from an EFSM by employing functional program testing [55]. The approach converted the specification written in Estelle [59] into a simpler form in order to construct control and data flow graphs to be used in test sequence derivation. However, these approaches did not consider the path feasibility problem.

Recent work has investigated the use of data flow when testing from UML statecharts [60]. Similar to [18], the focus was on defining and analysing dataflow, rather than on input sequence generation. The use of UML statecharts provides additional challenges because there is a need to analyse OCL guards. The authors then show how the result of dataflow analysis can be used to choose a transition tree: alternative transition trees provide different coverage of dataflow. The idea here is that there are alternative transition trees and the tester might choose between them on the basis of dataflow information. There is the potential to combine the dataflow analysis reported in [60] with the feasibility metric proposed in this paper in order to return a transition tree that has good dataflow coverage and whose paths are likely to be feasible.

Approaches that utilise search algorithms to test from EFSMs are introduced in [36, 56, 57]. The approach proposed in [57] describes a fitness calculation method to find an input sequence for a path. The considered fitness function applies the Tracey et al. [35] technique to each transition in a path. Path fitness is defined by considering each function in the path as a critical node. The limitation of this study is the assumption that each function does not have an internal path i.e. nested IF statements for which the Tracey et al. [35] approach does not always provide a sufficient guidance as argued in [5]. Furthermore, the work did not consider the problem of choosing a path that is likely to be feasible. In [36, 56] a GA was used to generate FTPs from an EFSM model. The approach evaluated the feasibility of a given TP according to the number and the types of guards found in that TP. However, the dependences between transitions in a path were not considered.

To summarise, while there has been significant interest in testing from an EFSM, there has been little work that uses search-based techniques. In addition, there has been almost no work that attempts to overcome the infeasible path problem in EFSMs and such work either has not been implemented or relies on the guards and operations being linear.

7. Conclusion

Although the EFSM is a powerful model and has been widely applied, testing from this model is a challenging task for two reasons: some paths may be infeasible and it may be difficult to produce an input sequence to execute a feasible transition path. Despite the fact that optimisation algorithms have proven to be effective in automating software testing, previously these have mainly been applied to white-box testing.

Many test criteria, for testing from an EFSM, require that certain parts of the EFSM are executed or reached in testing. Such criteria can be seen as placing requirements on the paths of the EFSM triggered by the chosen input sequences. For such criteria, one approach to finding a suitable set of input sequences is to first choose a set of transition paths that satisfies the criterion. It is important that the transition paths chosen are feasible but we also want to find input sequences to trigger them.

This paper addressed this problem by proposing an integrated search-based approach that has two phases. In the first phase a TP feasibility metric is used to guide a search towards paths that are likely to be feasible and that satisfy a give test criterion such as transition coverage. The second phase uses a fitness function that guides a search for an input sequence to exercise a given feasible TP.

We carried out experiments using five EFSMs with the aim of evaluating the proposed approach. A total of 363 transition paths were generated using the proposed TP feasibility metric. For each path, the proposed input sequence generator was applied to try to trigger it. Furthermore, we used a random approach to generate similar alternative TPs and also to generate input sequences to trigger the randomly generated TPs. Experimental results showed that the proposed feasibility metric successfully guided a GA search towards paths that are feasible with an accuracy rate of approximately 96.7%. The remaining 3.3% of paths were found to be infeasible due to a counter problem. Furthermore, the proposed input sequence generation method was found to be effective and successfully triggered all of the generated feasible paths (success rate = 100%). The results of the random approach showed that generating feasible transition paths from the considered EFSMs is not an easy task. Also, the random approach was not effective in generating input sequences to trigger the randomly generated FTPs. The overall success rate associated with the random approach was approximately 25.6% for FTPs generation and approximately 39.8% for input sequences generation.

We used a constraint based testing technique as an alternative method to generated input sequences to trigger the generated FTPs. Then, we performed a statistical analysis to investigate two questions: (1) the relationship between an FTP feasibility metric value and the effort, in terms of time[3], that is required to trigger the FTP. (2) Whether the feasibility metric can be used to predict such effort. Results show that there was a statistically significant correlation between the FTPs feasibility metric and the time that is required by both the proposed input sequence generator and the alternative constraint based input sequence generator. Interestingly, there was also a statistically significant agreement in the performance of the two input sequence generators although they used different techniques. Finally, the regression analysis showed that the proposed feasibility metric can potentially

[3] For the GA we looked at both GA time and number of generations but obtained almost identical results with these.

predict the effort by an input sequence generator to trigger an FTP. The feasibility metric was found to explain from 64% to 82% of the time variance.

Further research will investigate the scalability of the proposed approach by using different EFSM case studies. Furthermore, other penalty values will be used and the statistical study will be performed again to investigate whether the TP feasibility metric can reflect much higher correlations and prediction capability. Also, it would be interesting to investigate an FTP generation approach based on a dynamic analysis of the relations among an EFSM transitions. This would have the potential to possibly detect the counter problem and therefore generating FTPs that bypass this problem.

References:

[1]. Boehm, B.W., *Software Engineering Economics*. 1981, NJ: Prentice Hall PTR. 768.

[2]. Hierons, R.M., K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, *Using formal specifications to support testing.* ACM Computing Surveys, 2009. **41**(2): p. 1-76.

[3]. Korel, B., *Automated software test data generation.* Software Engineering, IEEE Transactions on, 1990. **16**(8): p. 870-879.

[4]. Michael, C.C., G. McGraw, and M.A. Schatz, *Generating software test data by evolution.* Software Engineering, IEEE Transactions on, 2001. **27**(12): p. 1085-1110.

[5]. McMinn, P., *Search-based software test data generation: a survey.* Software Testing, Verification & Reliability, 2004. **14**(2): p. 105-156.

[6]. Harman, M., L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, *Testability transformation.* Software Engineering, IEEE Transactions on, 2004. **30**(1): p. 3-16.

[7]. Petrenko, A., S. Boroday, and R. Groz, *Confirming configurations in EFSM testing.* Software Engineering, IEEE Transactions on, 2004. **30**(1): p. 29-42.

[8]. Lorenzoli, D., L. Mariani, and M. Pezz. *Automatic generation of software behavioral models*. in *30th international conference on Software engineering*. 2008. Leipzig, Germany: ACM Press.

[9]. Sinha, A., A. Paradkar, and C. Williams. *On Generating EFSM Models from Use Cases*. in *Sixth International Workshop on Scenarios and State Machines (SCESM '07)*. 2007: IEEE Press.

[10]. Dssouli, R., K. Saleh, E. M. Aboulhamid, A. En-Nouaary, and C. Bourhfir, *Test development for communication protocols: towards automation.* Computer Networks, 1999. **31**(17): p. 1835-1872.

[11]. Ural, H., K. Saleh, and A. Williams, *Test generation based on control and data dependencies within system specifications in SDL.* Computer Communications, 2000. **23**(7): p. 609-627.

[12]. Hierons, R.M., S. Sadeghipour, and H. Singh, *Testing a system specified using Statecharts and Z.* Information and Software Technology, 2001. **43**(2): p. 137-149.

[13]. Wong, W.E., A. Restrepo, and B. Choi, *Validation of SDL specifications using EFSM-based test generation.* Information and Software Technology, 2009. **51**(11): p. 1505-1519.

[14]. Keum, C., S. Kang, I.-Y. Ko, J. Baik, and Y.-I. Choi, *Generating Test Cases for Web Services Using Extended Finite State Machine*, in *Testing of Communicating Systems*. 2006. p. 103-117.

[15]. Tahat, L.H., A. Bader, B. Vaysburg, and B. Korel, *Requirement-based automated black-box test generation*. in *25th Annual International Computer Software and Applications Conference (COMPSAC '01)*. 2001: IEEE Press.

[16]. Duale, A.Y. and M.U. Uyar, *A method enabling feasible conformance test sequence generation for EFSM models.* Computers, IEEE Transactions on, 2004. **53**(5): p. 614-627.

[17]. Duale, A.Y., M. U. Uyar, B. D. McClure, and S. Chamberlain, *Conformance testing: towards refining VHDL specifications*. in *IEEE Military Communications Conference Proceedings (MILCOM 1999)*. 1999: IEEE Press.

[18]. Ural, H. and B. Yang, *A test sequence selection method for protocol testing.* Communications, IEEE Transactions on, 1991. **39**(4): p. 514-523.

[19]. Nilsson, R., J. Offutt, and J. Mellin, *Test Case Generation for Mutation-based Testing of Timeliness.* Electronic Notes in Theoretical Computer Science, 2006. **164**(4): p. 97-114.

[20]. Wenzel, I., R. Kirner, B. Rieder, and P. P. Puschner, *Measurement-Based Timing Analysis*, in

Leveraging Applications of Formal Methods, Verification and Validation. 2008, Springer: Berlin & Heidelberg. p. 430-444.
[21]. Lee, D. and M. Yannakakis, *Principles and methods of testing finite state machines-a survey.* Proceedings of the IEEE, 1996. **84**(8): p. 1090-1123.
[22]. Shih, C.-H., J.-D. Huang, and J.-Y. Jou. *Stimulus generation for interface protocol verification using the nondeterministic extended finite state machine model*. in *Tenth Annual IEEE International High-Level Design Validation And Test Workshop*. 2005: IEEE Press.
[23]. Tai, K.-C. *A program complexity metric based on data flow information in control graphs*. in *7th International Conference on Software Engineering (ICSE '84)*. 1984. Florida, US: IEEE Press.
[24]. Weiser, M. *Program slicing*. in *5th International Conference on Software Engineering (ICSE '81)*. 1981. San Diego, California, United States: IEEE Press.
[25]. King, J.C., *Symbolic execution and program testing.* Communications of the ACM, 1976. **19**(7): p. 385-394.
[26]. Harman, M. and B.F. Jones, *Search-based software engineering.* Information and Software Technology, 2001. **43**(14): p. 833-839.
[27]. Clark, J., J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper and M. Shepperd, *Reformulating software engineering as a search problem.* Software, IEE Proceedings -, 2003. **150**(3): p. 161-175.
[28]. Holland, J.H., *Adaptation in natural and artificial systems*. 1975, Ann Arbor: The University of Michigan Press. 211.
[29]. Harman, M. and J. Clark. *Metrics are fitness functions too*. in *10th International Symposium on Software Metrics (METRICS'04)*. 2004: IEEE Press.
[30]. Sadiq, M.S. and Y. Habib, *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems*. 1999, Los Alamitos, CA: IEEE. 387.
[31]. Srinivas, M. and L.M. Patnaik, *Genetic algorithms: a survey.* Computer, 1994. **27**(6): p. 17-26.
[32]. Kalaji, A.S., R.M. Hierons, and S. Swift. *Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)*. in *2nd IEEE International Conference on Software Testing, Verification, and Validation (ICST' 09)*. 2009. Denver, USA: IEEE Press.
[33]. Kalaji, A.S., R.M. Hierons, and S. Swift. *A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine (EFSM)*. in *Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC-PART)*. 2009. Windsor, UK: IEEE Press.
[34]. Wegener, J., A. Baresel, and H. Sthamer, *Evolutionary test environment for automatic structural testing.* Information and Software Technology, 2001. **43**(14): p. 841-854.
[35]. Tracey, N., J. A. Clark, K. Mander, and J. A. McDermid, *An automated framework for structural test-data generation*. in *13th IEEE International Conference on Automated Software Engineering*. 1998: IEEE Press.
[36]. Derderian, K., R. M. Hierons, M. Harman, and Q. Guo, *Estimating the feasibility of transition paths in extended finite state machines.* Automated Software Engineering, 2010. **17**(1): p. 33-56.
[37]. Tracey, N., J. Clark, and K. Mander. *Automated program flaw finding using simulated annealing*. in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1998. Clearwater Beach, Florida, United States: ACM Press.
[38]. Tracey, N., J. Clark, and K. Mander. *The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach*. in *The IFIP International Workshop on Dependable Computing and Its Applications (DCIA)*. 1998. South Africa.
[39]. Pohlheim, H. *GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab*. 1994-2010 [cited; Available from: http://www.geatbx.com.
[40]. Kalaji, A.S., R.M. Hierons, and S. Swift. *Generating Feasible Transition Paths for Testing from an Extended Finite State Machine with the Counter Problem*. in *3rd IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW' 10)*. 2010. Paris, France: IEEE Press.
[41]. Darringer, J.A. and J.C. King, *Applications of Symbolic Execution to Program Testing.* Computer, 1978. **11**(4): p. 51-60.
[42]. Matlab, *The Math Works- Optimization Toolbox:* *http://www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/fmincon.html*. 1984-2010.
[43]. Cohen, J., S*tatistical power analysis for the behavioral sciences*. 2nd ed. 1988, New Jersey: Lawrence

Erlbaum Associates.
[44]. Zhang, J., *Constraint Solving and Symbolic Execution*, in *Verified Software: Theories, Tools, Experiments*. 2008, Springer: Berlin / Heidelberg. p. 539-544.
[45]. Cheng, K.-T. and A.S. Krishnakumar, *Automatic generation of functional vectors using the extended finite state machine model.* ACM Transactions on Design Automation of Electronic Systems., 1996. **1**(1): p. 57-79.
[46]. Hierons, R.M., T.-H. Kim, and H. Ural, *On the testability of SDL specifications.* Computer Networks, 2004. **44**(5): p. 681-700.
[47]. Lee, D. and M. Yannakakis, *Testing finite-state machines: state identification and verification.* Computers, IEEE Transactions on, 1994. **43**(3): p. 306-320.
[48]. Dahbura, T.A., K.K. Sabnani, and M.U. Uyar, *Formal methods for generating protocol conformance test sequences.* Proceedings of the IEEE, 1990. **78**(8): p. 1317-1326.
[49]. Petrenko, A., G.v. Bochmann, and M. Yao, *On fault coverage of tests for finite state specifications.* Computer Networks and ISDN Systems, 1996. **29**(1): p. 81-106.
[50]. Koh, L.-S. and M.T. Liu. *Test path selection based on effective domains*. in *International Conference on Network Protocols (ICNP '94)*. 1994. Boston, MA: IEEE Press.
[51]. Bourhfir, C., R. Dssouli, and E.M. Aboulhamid, *Automatic Test Generation for EFSM-based Systems. Technical report*. 1996, University of Montreal, TR-1043. p. 1-59.
[52]. Zhang, J., C. Xu, and X. Wang. *Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques*. in *Second International Conference on Software Engineering and Formal Methods (SEFM '04)*. 2004: IEEE Press.
[53]. Chanson, S.T. and Z. Jinsong. *Automatic protocol test suite derivation*. in *13th IEEE Networking for Global Communications (INFOCOM '94)*. 1994: IEEE Press.
[54]. Chanson, S.T. and J. Zhu. *A unified approach to protocol test sequence generation*. in *12th Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future (INFOCOM '93)*. 1993: IEEE Press.
[55]. Sarikaya, B., G.v. Bochmann, and E. Cerny, *A Test Design Methodology for Protocol Testing.* Software Engineering, IEEE Transactions on, 1987. **SE-13**(5): p. 518-531.
[56]. Derderian, K., R. M. Hierons, M. Harman, and Q. Guo, *Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms*. in *The Genetic and Evolutionary Computation Conference (GECCO)*. 2005. Washington DC, USA: ACM Press.
[57]. Lefticaru, R. and F. Ipate. *Functional Search-based Testing from State Machines*. in *1st International Conference on Software Testing, Verification, and Validation (ICST '08)*. 2008: IEEE Press.
[58]. Hierons, R.M. and M. Harman, *Testing conformance of a deterministic implementation against a non-deterministic stream X-machine.* Theoretical Computer Science, 2004. **323**(1-3): p. 191-233.
[59]. Budkowski, S. and P. Dembinski, *An introduction to Estelle: a specification language for distributed systems.* Computer Networks and ISDN Systems., 1987. **14**(1): p. 3-23.
[60]. Briand, L., Y. Labiche, and Q. Lin, *Improving the coverage criteria of UML state machines using data flow analysis.* Software Testing, Verification and Reliability, 2010. **20**(3): p. 177-207.
[61]. Ramalingom, T., K. Thulasiraman, and A. Das, *Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines.* Computer Communications, 2003. **26**(14): p. 1622-1633.
[62]. Bochmann, G.V., *Specifications of a simplified transport protocol using different formal description techniques.* Computer Networks and ISDN Systems, 1990. **18**(5): p. 335-377.
[63]. Korel, B., L.H. Tahat, and B. Vaysburg. *Model based regression test reduction using dependence analysis*. in *International Conference on Software Maintenance (ICSM '02)*. 2002: IEEE Press.
[64]. Hogrefe, D., *OSI formal specification case study: the Inres protocol and service. Technical Report IAM-91-012*. 1991, University of Bern, Institute of Computer Science and Applied Mathematics. p. 5.
[65.] Goldberg, D. E., K. Deb, and J.H. Clark, *Genetic Algorithms, Noise, and the Sizing of Populations*, Complex Systems, 1992. 6: p. 333-362.
[66.] Carroll, D. L., *Chemical Laser Modeling with Genetic Algorithms*, AIAA J., 1996. 34(2): p. 338-346

Appendix A: Subject EFSMs

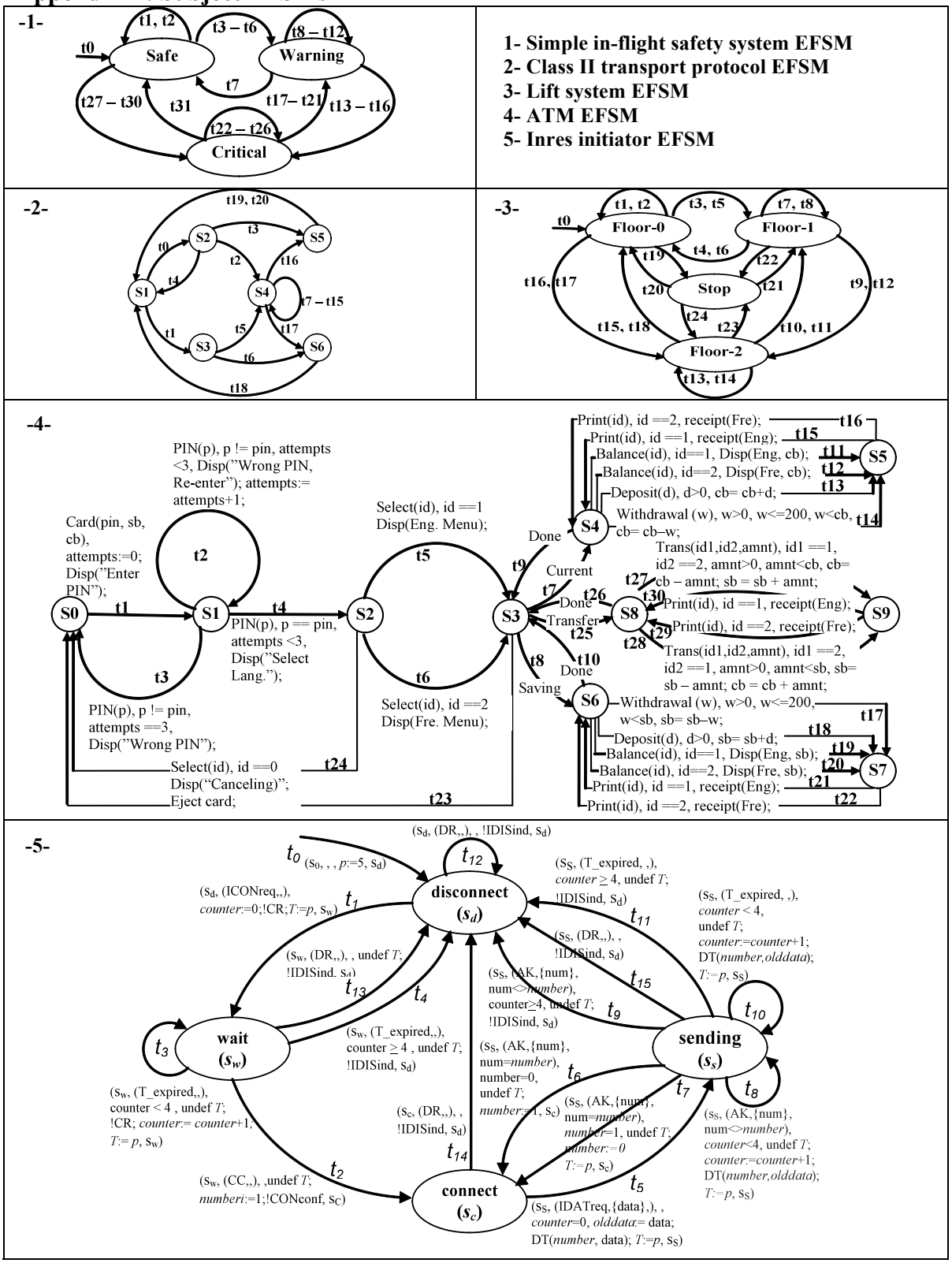


Figure 1. EFSM case studies

This appendix describes the five EFSM case studies (shown in Fig. A1) that are used in the empirical study to validate the proposed approach. In these five EFSMs, all the input parameters are of integer data type. When used, the symbol '?' indicates a request for an input whereas the Symbol "!" indicates an output. These EFSMs are:

1- **Simple in-flight safety system:** A synthesised simple system that functions as a monitor of the craft's cabin in terms of four factors: vibration, pressure, temperature and smoke. There are three states: (1) Safe when the values of these four factors are within a set of pre-defined ranges. (2) Warning when the value of one or more factors is within another set of pre-defined ranges. Here the pilot should take one or more actions according to a pre-defined list and the system can respond with some necessary actions i.e. when the air pressure is low, oxygen masks are released automatically. (3) Critical when the value of one or more factors is in a critical range and the pilot has to directly intervene. For example, if the pressure cannot be brought back to normal, an emergency landing might be taken. The EFSM has five context variables V= {VarsRead, Vb, Pr, Sm, Tm} and 31 transitions. Fig. A1-1 shows the EFSM and Table A1 lists the transitions specifications.

2- **Class II transport protocol:** This EFSM is based on the *AP-module* of the simplified version of a class 2 transport protocol. The EFSM model represents the core protocol transitions as described in [61] and [62]. This EFSM has two interaction points U and N for connecting to transport service access point and a mapping module respectively. The EFSM is involved in connection establishment, data transfer, end-to-end flow control and segmentation. This EFSM has seven states S = {s_0, s_1, s_2, s_3, s_4, s_5, s_6}, five context variables V = {opt, R-credit, S-credit, TRsq , TSsq} and 21 transitions. The model is shown in Fig. 1-2 and the transitions are described in Table A2.

3- **Lift system:** A synthesised lift system for a building with three floors. In order to open or close the lift cabin's door, the lift should be situated in the specified place within a margin that does not exceed 15%. The lift provides three operations: Request a lift from a specified floor, Service from a floor to another floor and Stop when there is a request. When a door is closed, the cabin load's weight is read and stored. In order for the cabin to move, the temperature and smoke level inside the cabin should be within pre-defined ranges. The lift does not provide a service if the cabin load is less than or equal to 15 KG so that a small child cannot operate the lift alone. The lift EFSM has four states S= {$Floor_0$, $Floor_1$, $Floor_2$, Stop}, three context variables V= {Drst, w, Floor} and 24 transitions. The EFSM is shown in Fig. A1-3 and the transitions are described in Table A3.

4- **ATM:** This represents an extension of the machine described in [63]. The machine offers the option of English or French menu and provides three services: Deposit, Withdrawal and Transfer between two accounts (Current and Saving). In order for a transaction to occur, a user must provide a valid PIN within three tries otherwise the machine will cancel the operation. The ATM EFSM has ten states S= {s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9}, four context variables V= {PIN, cb, sb, attempts} and 30 transitions. Fig. A1-4 shows the ATM EFSM and its transitions specifications.

5- Inres initiator: The Inres [64] protocol is connection-oriented and comprises the initiator, which establishes a connection and sends data, and the responder which receives data and terminates connections. The Inres protocol was designed to be similar to real protocols and yet small enough to allow experiments to be conducted for research purposes. The Inres initiator has five states S = {s_0, disconnect, wait, connect, sending}, four context variables V = {counter, number, T, p} and 15 transitions. Fig. A1-3 shows the Inres initiator EFSM together with the transitions specifications.

Table A1. The transitions specifications of the in-flight safety system EFSM

t $s_s \rightarrow s_e$	Input declarations	Guards	Transition atomic operations
t_0 $s_0 \rightarrow s_1$	reset	-	VarsRead= False; !SetWarningLights(all, off); !Sounds are switched off;
t_1 $s_1 \rightarrow s_1$ t_8 $s_2 \rightarrow s_2$ t_{22} $s_3 \rightarrow s_3$	?Read(Pvb, Ppr, Psm, Ptm)	VarsRead == False	Vb = Pvb; Pr = Ppr; Sm= Psm; Tm = Ptm; VarsRead = True;
t_2 $s_1 \rightarrow s_1$ t_7 $s_2 \rightarrow s_1$ t_{31} $s_3 \rightarrow s_1$	?MainCheck1 ()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 Tm ≥ 11 **&** Tm ≤ 35	VarsRead= False; !SetWarningLights(all, off); !Sounds are switched off;
t_3 $s_1 \rightarrow s_2$ t_9 $s_2 \rightarrow s_2$	?CheckVb1()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25	VarsRead= False; !SetLight(Seatbelt, on);
t_4 $s_1 \rightarrow s_2$ t_{10} $s_2 \rightarrow s_2$	?CheckPr1()	VarsRead == True **&** Pr ≥ 50 **&** Pr ≤ 85	VarsRead= False; Release(masks); !SetLight(Seatbelt, on);
t_5 $s_1 \rightarrow s_2$ t_{11} $s_2 \rightarrow s_2$	?CheckSm1()	VarsRead == True **&** Sm ≥ 11 **&** Sm ≤ 25	VarsRead= False; !SetSound(Sm, off);
t_6 $s_1 \rightarrow s_2$ t_{12} $s_2 \rightarrow s_2$	?CheckTm1()	VarsRead== True **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False; !SetLight(Tm, on);
t_{13} $s_2 \rightarrow s_3$ t_{23} $s_3 \rightarrow s_3$ t_{27} $s_1 \rightarrow s_3$	?CheckVb2()	VarsRead == True **&** Vb >25	VarsRead= False; !SetLight(Seatbelt, on);
t_{14} $s_2 \rightarrow s_3$ t_{24} $s_3 \rightarrow s_3$ t_{28} $s_1 \rightarrow s_3$	?CheckPr2()	VarsRead == True **&** Pr ≥ 0 **&** Pr ≤ 49	VarsRead= False; !Release(masks); !SetLight(Seatbelt, on); !SetSound(Pr, off);
t_{15} $s_2 \rightarrow s_3$ t_{25} $s_3 \rightarrow s_3$ t_{29} $s_1 \rightarrow s_3$	?CheckSm2()	VarsRead == True **&** Sm > 25	VarsRead= False !SetSound(Sm, off);
t_{16} $s_2 \rightarrow s_3$ t_{26} $s_3 \rightarrow s_3$ t_{30} $s_1 \rightarrow s_3$	?CheckTm2()	VarsRead= True **&** (Tm >46) **V** (Tm ≤2)	VarsRead= False !SetLight(Tm, on); !SetLight(AC, on);
t_{17} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25 **&** Pr ≥ 50 **&** Pr ≤ 85 **&** Sm ≥ 11 **&** Sm ≤ 25 **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False !SetWarningLights(all, on); !SetWarningSounds (all, off); !Release(masks);
t_{18} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 11 **&** Vb ≤25 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !SetLight(Seatbelt, on);
t_{19} s3 $\rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 50 **&** Pr ≤ 85 **&** Sm ≥0 **&** Sm ≤ 10 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !Release(masks); !SetLight(Seatbelt, on); !SetSound(Pr, off);
t_{20} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥ 11 **&** Sm ≤ 25 **&** Tm ≥ 11 **&**Tm ≤ 35	VarsRead= False; !SetSound(Sm, off);
t_{21} $s_3 \rightarrow s_2$	?MainCheck2()	VarsRead == True **&** Vb ≥ 0 **&** Vb ≤10 **&** Pr ≥ 86 **&** Pr ≤ 100 **&** Sm ≥0 **&** Sm ≤ 10 **&** (Tm ≥ 36 **&** Tm ≤ 46) **V** (Tm ≥ 3 **&** Tm ≤ 10)	VarsRead= False !SetLight(Tm, on); !SelLight(AC, on);

Table A2. The core transitions in the class II transport protocol EFSM

t $s_s \rightarrow s_e$	Input declarations	Guards	Transition atomic operations
t_0 $s_1 \rightarrow s_2$	U?TCONreq(dst_add, prop_opt)	-	opt = prop_opt; R_credit =0; N!TrCR
t_1 $s_1 \rightarrow s_3$	N?TrCR(peer_add, opt_ind, cr)	-	opt= opt_ind; S_credit=cr; R_credit=0; U!TCONind
t_2 $s_2 \rightarrow s_4$	N?TrCC(opt_ind, cr)	opt_ind < opt	TRsq=0; TSsq=0; opt=opt_ind; S_credit=cr; U!TCONconf
t_3 $s_2 \rightarrow s_5$	N?TrCC(opt_ind, cr)	opt_ind > opt	U!TDISind; N!TrDR
t_4 $s_2 \rightarrow s_1$	N?TrDR(disc_reason, switch)	-	U!TDISind; N!terminated
t_5 $s_3 \rightarrow s_4$	U?TCONresp(accpt_opt)	accpt_opt < opt	opt= accpt_opt; TRsq=0; TSsq=0; N!TrCC
t_6 $s_3 \rightarrow s_6$	U?TDISreq()	-	N!TrDR
t_7 $s_4 \rightarrow s_4$	U?TDATAreq(Udata, E0SDU)	S_credit > 0	S_credit= S_credit -1; TSsq = (TSsq +1)mod128; N!TrDT
t_8 $s_4 \rightarrow s_4$	N?TrDT(Send_sq, Ndata, E0TSDU)	R_credit != 0 **&** Send_sq== TRsq	TRsq=(TRsq+1)mod128; R_credit=R_credit -1; U!DATAind; N!TrAK
t_9 $s_4 \rightarrow s_4$	N?TrDT(Send_sq, Ndata, E0TSDU)	R_credit == 0 **V** Send_sq != TRsq	U!error; N!error
t_{10} $s_4 \rightarrow s_4$	U?U READY(cr)	-	R_credit= R_credit + cr; N!TrAK
t_{11} $s_4 \rightarrow s_4$	N?TrAK(XpSsq, cr)	TSsq > XpSsq **&** cr + XpSsq – TSsq ≥ 0 **&** cr +XpSsq – TSsq ≤ 15	S_credit = cr + XpSsq – TSsq
t_{12} $s_4 \rightarrow s_4$	N?TrAK(XpSsq, cr)	TSsq ≥ XpSsq **&** (cr + XpSsq – TSsq < 0 **V** cr +XpSsq – TSsq >0**)**	U!error; N!error
t_{13} $s_4 \rightarrow s_4$	N?TrAK(XpSsq, cr)	TSsq < XpSsq **&** cr + XpSsq – TSsq – 128 ≥ 0 **&** cr + XpSsq – TSsq – 128 ≤ 15	S_credit= cr+ XpSsq –TSsq – 128
t_{14} $s_4 \rightarrow s_4$	N?TrAK(XpSsq, cr)	TSsq < XpSsq **&** (cr + XpSsq – TSsq – 128 < 0 **V** cr + XpSsq – TSsq – 128 > 15 **)**	U!error; N!error
t_{15} $s_4 \rightarrow s_4$	N?Ready()	S_creidit > 0	U!Ready
t_{16} $s_4 \rightarrow s_5$	U?TDISreq()	-	N!TrDR
t_{17} $s_4 \rightarrow s_6$	N?TrDR(disc_reason, switch)	-	U!TDISind; N!TrDC
t_{18} $s_6 \rightarrow s_1$	N?terminated()	-	U!TDISconf
t_{19} $s_5 \rightarrow s_1$	N?TrDC()	-	N!terminated; U!TDISconf
t_{20} $s_5 \rightarrow s_1$	N?TrDR(disc_reason, switch)	-	N!terminated

Table A3. The transitions specifications of the Lift system EFSM

t $s_s \rightarrow s_e$	Input declarations	Guards	Transition atomic operations
$t_0 \rightarrow s_0$	reset		Floor = 0; DrSt = 0; w = 0;
t_1 $s_0 \rightarrow s_0$	?DrOp(Pos)	DrSt == 0 & Pos ≥ 0 & Pos ≤15	DrSt = 1;
t_2 $s_0 \rightarrow s_0$	?DrCl(Pos, Pw)	DrSt == 1 & Pos ≥ 0 & Pos ≤15	DrSt = 0; w = Pw
t_3 $s_0 \rightarrow s_1$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 1 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 1; !Display(Floor);
t_4 $s_1 \rightarrow s_0$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 0 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 0; !Display(Floor);
t_5 $s_0 \rightarrow s_1$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf == 1 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 1; !Display(Floor);
t_6 $s_1 \rightarrow s_0$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf == 0 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 0; !Display(Floor);
t_7 $s_1 \rightarrow s_1$	?DrOp(Pos)	DrSt == 0 & Pos ≥ 0 & Pos ≤15	DrSt = 1;
t_8 $s_1 \rightarrow s_1$	?DrCl(Pos, Pw)	DrSt == 1 & Pos ≥ 0 & Pos ≤15	DrSt = 0; w = Pw
t_9 $s_1 \rightarrow s_2$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 2 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 2; !Display(Floor);
t_{10} $s_2 \rightarrow s_1$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 1 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 1; !Display(Floor);
t_{11} $s_2 \rightarrow s_1$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf ==1 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 1; !Display(Floor);
t_{12} $s_1 \rightarrow s_2$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf == 2 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 2; !Display(Floor);
t_{13} $s_2 \rightarrow s_2$	?DrOp(Pos)	DrSt == 0 & Pos ≥ 0 & Pos ≤15	DrSt = 1;
t_{14} $s_2 \rightarrow s_2$	?DrCl(Pos, Pw)	DrSt == 1 & Pos ≥ 0 & Pos ≤15	DrSt = 0; w = Pw
t_{15} $s_2 \rightarrow s_0$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 0 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 0; !Display(Floor);
t_{16} $s_0 \rightarrow s_2$	?Srv(Pf, Ph, Ps)	DrSt == 0 & Pf == 2 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 2; !Display(Floor);
t_{17} $s_0 \rightarrow s_2$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf ==2 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 2; !Display(Floor);
t_{18} $s_2 \rightarrow s_0$	?Req (Pf, Ph, Ps)	DrSt == 0 & Pf == 0 & w =0 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 0; !Display(Floor);
t_{19} $s_0 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 & Pf == 100 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 100; !Display(Floor);
t_{20} $s_s \rightarrow s_0$	?Srv(Pf)	DrSt == 0 & Pf == 0	Floor = 0; !Display(Floor);
t_{21} $s_s \rightarrow s_1$	?Srv(Pf)	DrSt == 0 & Pf == 1	Floor = 1; !Display(Floor);
t_{22} $s_1 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 & Pf == 100 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 100; !Display(Floor);
t_{23} $s_2 \rightarrow s_s$	?Stp (Pf, Ph, Ps)	DrSt == 0 & Pf == 100 & w ≥15 & w ≤ 250 & Ph ≥ 10 & Ph ≤ 35 & Ps ≥ 0 & Ps ≤25	Floor = 100; !Display(Floor);
t_{24} $s_s \rightarrow s_2$	?Srv(Pf)	DrSt == 0 & Pf == 2	Floor = 2; !Display(Floor);