2009 International Conference on Software Testing Verification and Validation

Generating Feasible Transition Paths for Testing from an Extended Finite State Machine (EFSM)

AbdulSalam Kalaji, Robert M Hierons and Stephen Swift
School of Information Systems, Math and Computing
Brunel University, Uxbridge, UB8 3PH, UK
{abdulsalam.kalaji, rob.hierons, stephen.swift}@brunel.ac.uk

Abstract

The problem of testing from an extended finite state machine (EFSM) can be expressed in terms of finding suitable paths through the EFSM and then deriving test data to follow the paths. A chosen path may be infeasible and so it is desirable to have methods that can direct the search for appropriate paths through the EFSM towards those that are likely to be feasible. However, generating feasible transition paths (FTPs) for model based testing is a challenging task and is an open research problem. This paper introduces a novel fitness metric that analyzes data flow dependence among the actions and conditions of the transitions of a path in order to estimate its feasibility. The proposed fitness metric is evaluated by being used in a genetic algorithm to guide the search for FTPs.

1. Introduction

Errors in software and hardware can cause undesired consequences and testing is therefore an important stage of the software development process. However, manual testing is expensive, error-prone and time consuming hence automation is very desirable. Automated test data generation has been a subject of interest for many researchers in the last decade, the objective being to develop efficient methods which can replace conventional manual methods [1-3].

When a system is implemented, it is necessary to test whether the implementation agrees with its specification. This is usually performed by conducting conformance testing which tries to find behaviours of an implementation under test (IUT) that are not consistent with the specification. In order to derive a test sequence from a system specification, a model that represents the specification is required. Finite state machines (FSMs) and extended finite state machines (EFSMs) are commonly used in test sequence derivation [4]. An FSM can only model the control part of a system; an extension is needed in order to model a system which has control and data parts. Such systems are usually represented by using an EFSM model.

The FSM model has been widely studied and many methods are available for the purpose of test data generation [5, 6]. Nevertheless, automated test data generation from an EFSM model is complicated by the presence of infeasible paths and is an open research problem [7]. When testing from an EFSM, there are several test strategies such as state coverage, transition coverage and path coverage that require the generation of a set of FTPs in order to produce a test suite [8].

In an EFSM model, a given transition path (TP) can be infeasible due to the variable interdependencies among the actions and conditions. If an infeasible path is chosen to exercise certain transitions, these transitions are not exercised. Problems arising from the existence of infeasible paths are generally undecidable [9]. In addition, feasible transition paths are subject to different levels of traversal complexities; it can be hard to find a set of test data that can trigger a given FTP.

A path test data is a sequence of input values to be applied to the interaction parameter fields of the transitions included in that path. Generally, finding a suitable sequence of input values to trigger an FTP in an EFSM is a hard task [10]. If test data is to be produced from an EFSM model by first deriving a set of TPs then it is necessary to use feasible TPs. It is also desirable that these FTPs are relatively easy to trigger but still satisfy the test criterion being used.

The motivation for our work is the observation that when generating a set of paths to satisfy a test criterion there are many alternative choices and we would like to produce one of the sets that contains paths that are feasible and relatively easy to trigger. We are particularly interested in search based test generation and so the aim is to produce a fitness metric that can be computed quickly and that can be used as part of an overall fitness function. In particular, if we have a

978-0-7695-3601-9/09 $25.00 © 2009 IEEE
DOI 10.1109/ICST.2009.29

fitness function that directs search towards paths that satisfy a current test objective (part of a test criterion) then we can see the problem of producing an appropriate path as a multi-objective search problem. Ultimately, we believe that fitness metrics such as ours can be incorporated into the search when using any available testing technique that require the generation of a set of feasible paths through an EFSM model to satisfy a particular test criterion [4, 6, 7, 9, 11-17]

Since we are interested in producing a fitness metric that can be used in search based testing we want our fitness metric to direct the search towards paths that are relatively easy to trigger if we then use search to find test data. As a result, we measure how easy it is to trigger a path using search by estimating how easy it is to find test data to trigger the path using random test data generation. To this end, the approach presented in this paper aims to form part of the solution to the following problem:

Given: an EFSM model and a test adequacy criterion
Problem: generate a set of TPs that are feasible, are easy to trigger, and satisfy the test criterion.

The primary contributions of this paper are the following:

1. It proposes a method that enables automatic generation of FTPs from EFSMs models.
2. It shows how the proposed fitness metric can make the process of path test data generation easier through producing FTPs that are relatively easy to trigger.
3. The paper empirically validates the efficiency of the proposed fitness metric by using it with the class 2 transport protocol EFSM.

The rest of the paper is organized as follows: Section 2 provides background information and an overview of related test generation methods. The proposed approach is described in Section 3. Experimental results are discussed in Section 4. Concluding remarks and future work are in Section 5.

2. Preliminaries

2.1. The model

The EFSM model is a 6-tuple [15] *(S, s_0, V, I, O, T)* where *S* is a finite set of logical states, $s_0 \in S$ is the initial state, *V* is a finite set of internal variables, *I* is a set of input declarations, *O* is the set of output declarations and *T* is a 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 such as output or assignment statements and s_e is the end state of *t*.

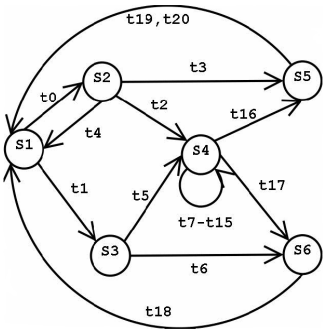

Figure 1. Class 2 transport protocol EFSM

In an EFSM model, there is a set of variables. One variable in particular is used to represent the machine state[1] and is called state or major state in order to differentiate it from the other variables called context variables. The state variable is used to represent the state of a finite state machine i.e. idle, wait for connection and so on, whereas other machine data such as port number and sequencing numbers are usually stored in context variables. A state transition occurs when one of the machine's transitions is taken. Each transition has two major states: start state (s_s) and end state (s_e). If a transition *t* has a guard *g* on the context variables and input parameters then *g* must be satisfied in order for *t* to be taken. Also, a transition *t* may have one or more atomic operations (op) to be executed when *t* is taken [16]. 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 [18]. In this paper, we only consider deterministic EFSMs.

2.2. A case study

In this paper, we present an EFSM case study that we use to illustrate and validate the proposed approach. The EFSM is a major model based on the *Access Point* (*AP)-module,* described in [19], of the simplified version of a class 2 transport protocol. The EFSM model represents the core protocol transitions as described in [15] and [19]. This protocol has two interaction points U and N for connecting to a transport service access point and a mapping module respectively. The protocol is involved in connection establishment, data transfer, end-to-end flow control and segmentation. The input declarations are given in terms of an input signal, which is identified by using '?', and the required input parameters. Output signals

[1] Naturally, the state variable may represent a tuple of values.

Table 1. Core transitions in the class 2 transport protocol

t $s_s \rightarrow s_e$	Input declarations and Params	Transition guards	Transition atomic operations
t_0 $s_1 \rightarrow s_2$	U?TCONreq (dst_add, prop_opt)	Nil	opt = prop_opt; R_credit =0; N!TrCR
t_1 $s_1 \rightarrow s_3$	N?TrCR (peer_add, opt_ind, cr)	Nil	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)	Nil	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 ()	Nil	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)	Nil	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_credit > 0	U!Ready
t_{16} $s_4 \rightarrow s_5$	U?TDISreq	Nil	N!TrDR
t_{17} $s_4 \rightarrow s_6$	N?TrDR (disc_reason, switch)	Nil	U!TDISind; N!TrDC
t_{18} $s_6 \rightarrow s_1$	N?terminated	Nil	U!TDISconf
t_{19} $s_5 \rightarrow s_1$	N?TrDC	Nil	N!terminated; U!TDISconf
t_{20} $s_5 \rightarrow s_1$	N?TrDR (disc_reason, switch)	Nil	N!terminated

are identified by using '!'. In this EFSM, all the variables and the input parameters are of integer data type. We exclude some input parameters {dst_add, peer_add, E0TSDU, disc_reason, switch} since they are not involved in any guards or assignments. The transitions are shown in Fig.1 and described in Table 1.

2.3. Genetic algorithms

Genetic Algorithms (GAs) are powerful, simple, and sturdy heuristic techniques that implement the natural selection theory [20] and have been widely applied to solve optimization problems. GAs work on a set of candidate solutions called a population. GAs require a suitable representation of candidate solutions in order for them to be applied to a particular problem. This can be achieved by using a solution encoding such as binary or real valued encoding. Each encoded solution is called a chromosome and each chromosome has one or more components which are called genes [21].

The GA's cycle starts by evaluating the fitness of each individual, this fitness being a positive value that measures how 'fit' it is and hence its chance of use as a parent. Then a selection based on fitness such as Roulette wheel or ranking [22] is made to perform 'breeding'. Through 'breeding' new individuals are introduced. This is accomplished by applying a crossover operator. Crossover or recombination acts on two individuals to produce two new individuals and can be performed in several ways. The simplest one is referred to as one-point crossover. It operates by choosing a random position on the chromosome's bit string, and then the substrings before that position are kept while the tails are swapped [23]. For example, Pa_1 and Pa_2 are two parents recombined at position 4 to produce the offspring C_1 and C_2:

$$Pa_1\ \{011|00\} \qquad C_1\ \{011|11\}$$
$$Pa_2\ \{101|11\} \qquad C_2\ \{101|00\}$$

In order to maintain population diversity, new characteristics are infrequently injected by applying mutation. Mutation acts on one chromosome at a time and randomly changes the values of some of its genes [23]. For example, C_1 above might become C_1' after mutating the genes on positions 1 and 5.

$$C_1\ \{01111\} \qquad C_1'\ \{11110\}$$

The GA's cycle usually yields 'fitter' individuals that are used to replace or update the population. There is a sequence of updates until it satisfies one of the stopping criteria such as finding a solution or reaching a maximum number of generations [24].

2.4. Definition-use and data flow dependence

Given a variable x within a program, x is said to be defined at the program node n_1 when the statement at

n_1 assigns a value to x. A use of x occurs when an assignment or output statement at the program node n_2 references x (c-use) or a conditional statement references x (p-use). If the definition of x at n_1 propagates to be used at n_2 and x is only defined at n_1 and not redefined before reaching n_2 then the path from n_1 to n_2 is definition clear for x and (n_1, n_2) forms a definition-use (*du)* pair for x [25]. When the path from n_1 to n_2 is a *du* for x, then there is data flow dependence between n_1 and n_2 [26].

2.5. Related work

There are many test generation approaches for systems modeled as EFSMs [7, 9, 11, 13, 16, 27-29]. An approach to generate a unified test sequence (UTS) for EFSM models is presented in [9]. The approach is based on two techniques: one to test the control part (FSM) and the other to test the data part by using data flow analysis. The resultant UTS is then checked for executability by using a constraint satisfaction method. However, the assumption about the existence of the self-loop influencing (a loop that modifies a global predicate variable) in the considered EFSM may not hold for all EFSMs.

Generating test sequences for EFSM models by employing functional program testing is studied in [16]. The approach converts an Estelle specification into a simpler form in order to construct control and data flow graphs to be used in test sequence derivation in a non-automatic fashion.

Other methods that test from an EFSM model using FSM-based test techniques appear in [27-29]. Generally, the notion of testing from EFSM models based on FSM methods requires a transformation from EFSM to FSM. There are two approaches, the first being to abstract the data from an EFSM model. The limitation of this approach is that the paths taken from the FSM model are not necessarily feasible in the EFSM. The second approach is to expand an EFSM to form an FSM, however, the number of states in the resultant FSM can become prohibitively large [30].

A method that overcomes the feasibility problem in advance is introduced in [7]. The approach requires that all the conditions and actions in the EFSM are linear and converts a class of EFSMs into EFSMs in which all TPs are feasible. However, the approach does not provide an assessment to the traversal complexity associated with generated FTPs and so an FTP can be feasible but still very hard to trigger.

A technique to bypass the infeasible path problem in an EFSM is presented in [13]. First, the specifications are rewritten in order to derive a normal form EFSM (NF-EFSM). Then, the NF-EFSM is extended to Expanded-EFSM (EEFSM) with the property that all the paths in the EEFSM are feasible. However, the study does not provide an assessment of the traversal complexity of the resultant FTPs.

A GA approach to generate FTPs from EFSM model is presented in [11]. This is the only previous work that utilizes a GA to generate FTPs. The approach evaluates the feasibility of a given TP according to the number and the types of guards found in that TP. However, the transition interdependencies are not considered in assessing a given TP feasibility.

3. Proposed FTPs generation approach

Although the techniques described in Section 2 made considerable contributions towards EFSM testing, they have not solved the problem of generating a set of TPs that can be easily triggered using search. Before giving a detailed description of our approach, we introduce some definitions:

Definition 1: A transition path TP of length n is a sequence of n consecutive transitions $t_1, t_2, .., t_n$.

Definition 2: A TP is said to be an FTP *iff* it is possible to trigger each transition t_i, $1 \leq i \leq n$, and in the sequential order that it appears in this TP.

A transition's guard has the form *(e gop e')* where e and e' are expressions and $gop \in \{<, >, \neq, =, \leq, \geq\}$ is the guard operator. Given an expression e, we let *Ref(e)* denote the set of variables that appear in this expression. According to e and e' a transition's guard can be classified into the following types:

1. g^{pv}: a comparison involving a parameter and zero or more context variables; there exists a parameter $p \in Ref(e) \cup Ref(e')$.
2. g^{vv}: a comparison among context variables' values; every element of $Ref(e) \cup Ref(e')$ is a context variable.
3. g^{vc}: a comparison between a constant and an expression involving context variables' values; all elements of $Ref(e) \cup Ref(e')$ are context variables and either e or e' is a constant.

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

1. op^{pv}: it assigns to v a value that depends on the parameter and so there is a parameter $p \in Ref(e)$.
2. op^{vv}: it assigns to v a value that depends only on the context variable(s) and so all the elements of *Ref(e)* are context variables.
3. op^{vc}: it assigns to v a constant value and so e is a constant.

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

Definition 3: In a given TP $t_1, t_2, .., t_n, t_i$ is an affecting transition within this TP *iff* t_i has an assignment $op \in \{op^{pv}, op^{vc}, op^{vv}\}$ 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 the same v and the path from t_i to t_j is definition clear for v.

Definition 4: In a given TP $t_1, t_2, .., t_n, t_j$ is an affected-by transition *iff* t_j has a guard $g \in \{ g^{pv}, g^{vv}, g^{vc}\}$ over v and there exists an affecting transition $t_i \in$ TP, where $1 \leq i < j \leq n$ over the same v and the path from t_i to t_j is definition clear for v.

Definition 5: An assignment op of the type op^{vc} is opposed to a guard g of the type g^{vc} when either the constants that appear in op^{vc} and g^{vc} are the same and $gop \in \{<, >, \neq\}$ or are different and $gop \in \{=\}$ and the path from op to g is definition clear for the variable v that appears in op^{vc} and g^{vc}.

According to Definitions 3, 4 and 5, we can define a case where a given TP is clearly infeasible:

Infeasible TP: A given TP $t_1, t_2, .., t_n$ with length $n > 1$ is definitely infeasible if there exists a variable v and a pair of transitions (t_i, t_j) where $1 \leq i < j \leq n$, t_i is an affecting transition of type op^{vc}, t_j is an affected-by transition of type g^{vc} and op^{vc} opposes g^{vc}.

3.1. Dependencies representation and penalties

In order to estimate the fitness of a given TP, we require an analysis of all the dependencies among the affecting and affected-by transitions in that TP. In order to have a fitness metric that can be computed quickly we base the path fitness assessment on an approximate value calculated by using penalty values that are determined in advance, thus each calculation is computationally simple and can be used in search algorithms. There are three factors that need to be considered when assigning a penalty value to a pair of affected-by and affecting transitions. First, the guard type that occurs in an affected-by transition determines how much this transition can be impacted by another affecting transition. For example, a guard of type g^{pv} is considered the easiest guard to satisfy since we can try to choose the parameter value in order to satisfy this guard. Thus an affecting transition's assignment does not play such an important role in worsening this guard complexity. A guard of type g^{vc} is considered the hardest since we cannot set the value of either c or v involved in this guard. Secondly, the guard operator can impact the guard complexity. For example, it is usually easier to satisfy a guard with inequality than a guard with equality. Finally, the assignment type of an affecting transition can improve the complexity of an affected-by guard when a parameter value is involved but has a negative impact when a constant value is involved. For example, consider an affected-by transition with a $g^{vc}(=)$ guard and an affecting transition with an (op^{vc}) assignment. Unless the constant is the same in both transitions, the corresponding TP is infeasible. Table 2 shows the suggested penalty values for all possible combinations of guards and assignments between a pair of affected-by and affecting transitions.

Naturally, guards can contain predicates linked with 'OR' and 'AND' operators. In such cases, we consider the smallest penalty value when the 'OR' operator is used. Alternatively, it is possible to split the transition into a number of transitions equal to the number of 'OR' operator used + 1. For predicates linked with 'AND', the sum of penalty values is used.

Any possible dependency between a pair of affecting and affected-by transitions includes at least one of the EFSM's context variables and an affected-by transition in a given TP can depend on several affecting transitions. In a preprocessing step, we record the type of dependency that occurs at each context variable for each pair of transitions. From an affecting transition's assignments, three types of dependencies can be distinguished where we represent each assignment type as a unique integer value. The negative values of {-1, -2} represent the assignment to a parameter (op^{pv}) and a constant (op^{vc}) respectively whereas the positive values of {1, 2, ..., m} represent the assignment to a context variable (op^{vv}) where the referenced context variable is represented by its integer index. If an assignment of the type (op^{vv}) contains an expression which references more than one context variable, then only one of the referenced context variables v' is considered according to the following order that depends on the previous assignment to v': (1) the previous assignment to v' references a parameter. (2) the previous assignment to v' references a constant. (3) v' has the fewest assignments that affect its value.

In addition to these three types of dependencies, there may be no dependency or an open ended dependency (when a variable references another variable(s) which, in turns, is left undefined). We represent this type as 0. Table 3 lists all the types of dependency together with their integer representation.

Example 1. The EFSM shown in Fig.1 has five context variables {*opt*, *R_credit*, *S_credit*, *TRsq*, *TSsq*} which we will refer to henceforth by $\{v_1, v_2, v_3, v_4, v_5\}$ respectively. Let us consider transitions t_2 and t_7; from Table 1, t_7 is an affected-by transition of type $g^{vc}(>)$ whereas t_2 is an affecting transition of type op^{pv}. From Table 2, the associate penalty value is 24. The

Table 2. The suggested penalty values

Guard & operator	Assignment (op^{pv})	(op^{vv})	(op^{vc})
$g^{pv}(=)$	8	16	24
$g^{pv}(<, >)$	6	12	18
$g^{pv}(\leq, \geq)$	4	8	12
$g^{pv}(\neq)$	2	4	6
$g^{vv}(=)$	20	40	60
$g^{vv}(<, >)$	16	32	48
$g^{vv}(\leq, \geq)$	12	24	36
$g^{vv}(\neq)$	8	16	24
$g^{vc}(=)$	30	60	500 if c is different and 0 otherwise
$g^{vc}(<, >)$	24	48	0 if c is different and 500 otherwise
$g^{vc}(\leq, \geq)$	18	36	500 if c is different and 0 otherwise
$g^{vc}(\neq)$	12	24	0 if c is different and 500 otherwise

dependency between these transitions occurs only through the context variable v_3. Thus, we can formulate this dependency as a tuple with six fields. Five of them represent the dependencies between the two transitions that occur at all context variables, whereas the last one is used to record the associated penalty value:

			t_2			
t_7	$v_1 = 0$	$v_2 = 0$	$v_3 = -1$	$v_4 = 0$	$v_5 = 0$	Penalty = 24

The information in the above tuple can be interpreted by using Table 3 as: there is a dependency between the two transitions at the context variable v_3 and this dependency ends with an assignment of a parameter value to v_3.

All the dependencies among transitions can be represented in terms of an array with two dimensions. In this array, rows represent affected-by transitions whereas columns represent affecting transitions. Each cell of this array has the form of the above mentioned tuple. In the experiment, we constructed this array manually, however, it is possible to derive this array automatically.

Example 2. The following array represents the analysis information for all the dependencies among the affecting transition t_0, t_1, t_2 and t_3 and the affected-by transitions t_0, t_1, t_2, t_3, t_4 and t_5.

	t_0						t_1						t_2						t_3					
t_0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t_2	-1	0	0	0	0	4	-1	0	0	0	0	4	-1	0	0	0	0	4	0	0	0	0	0	0
t_3	-1	0	0	0	0	6	-1	0	0	0	0	6	-1	0	0	0	0	6	0	0	0	0	0	0
t_4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t_5	-1	0	0	0	0	4	-1	0	0	0	0	4	-1	0	0	0	0	4	0	0	0	0	0	0
	v_1	v_2	v_3	v_4	v_5	→ Penalty																		

3.2. The fitness metric

In this subsection we define the proposed fitness metric. Fig.2 shows a high level description of the algorithm that calculates the fitness metric. Given a TP with length $n > 1$, the algorithm computes the fitness

Table 3. Assignment's types representation

op	Representation	Meaning
op^{pv}	-1	A dependency ends with a parameter value
op^{vc}	-2	A dependency ends with a constant value
op^{vv}	$v_{1..m}$	A dependency continues by referencing context variables
none	0	No dependency or open ended dependency

by starting from the last transition t_n and checking the dependencies (if any) with the prior transition t_{n-1}. From the array that contains the analysis information, if the transitions (t_n, t_{n-1}) form a pair of affected-by and affecting transitions then the associated penalty value is different from 0 (Line 11). In this case, the algorithm determines the context variable at which the dependency has occurred (Line 14). There are two cases: first, if the dependency ends with an assignment to a parameter or a constant value (Line 16) then the associated penalty value of (t_n, t_{n-1}) is considered only. In the second case, the dependency continues as a reference to one or more context variables. At this point, a call is made to a recursive function *check* which traces back all the previous assignments which are propagated to the context variable v. The function *check* performs backward data flow dependence analysis for variable v at transition t_n where the function stops when it reaches an assignment to a parameter or constant value. That is, for a context variable v, we want to know how many assignments contribute to the value of this variable (Line A15) by starting from the current transition (Line A10) and working backwards to all prior transitions. A value of 40 is added each time an assignment that contributes to the value of v is found. When v is eventually, directly or indirectly, defined by a parameter value (Line A14), a penalty value of 20 is added, however, if the dependency is ended by referring to a constant value (Line A13) or left open ended (Line A22) then a penalty value of 60 is added. The total penalty value calculated by *check* is passed back to the main function to be added to the final result. Once all dependencies between the transition t_n and previous transitions are detected, a new loop cycle starts to evaluate the next pair of transitions (t_{n-1}, t_{n-2}) and so forth.

This fitness metric penalizes complexity (there being many assignments affecting the variables in a guard). It also penalizes certain features of guards, such as comparing two expressions only containing context variables using =, that are likely to lead to them being difficult to satisfy. The aim is thus to penalize factors that could make it hard to find test data to trigger the path but naturally this cannot be entirely precise. The fitness metric is simple to compute and so can be incorporated into search based testing where we may have to evaluate fitness hundreds of thousands of times. We did not explore the effect of different

A TP fitness metric

```
1. begin
2. input: TP, EFSM analysis array
3. output: non negative integer value
4. goal: evaluate a TP complexity
5. initialize variable result = 0;
6. for i = n downto first_transition
7.  begin
8.   j= i;
9.    repeat
10.     j = j -1;
11.     if [ti,tj].penalty > 0 then
    // if these Trans. form a pair of affected-by & affecting Trans.
12.       begin
13.         result = result + [ti,tj].penalty ;
    // add the associated penalty value
14.         for v = v1 to vk do
    // check at which context Var. the dependency occurs
15.          begin
16.           if [ti,tj].v < 0 then continue;
    // the dependency ends by a Param. or a constant value
17.           if [ti,tj].v > 0 then
    // the dependency continues by referencing a context Var.
18.            result = result + check(ti,tj,v);
    // call check function to trace back all the dependencies
    // that propagated at this context Var.
19.          end;
20.       end;
21.    until j = first_transition
22.  end;
23. return result;
24. end
```

Function *check* all of a transition dependencies

```
A1. begin
A2. input: ti,tj,v
A3. output: non negative integer value
A4. goal: trace back a flow dependence on variable v
A5. initialize variable result = 0; found = false;
A6. begin
A7.   p = j + 1;
A8.   repeat
A9.     p = p – 1;
A10.    if [ti,tp].v ≠ 0 then
A11.     begin
A12.       case [ti,tp].v of
      // check the type of dependency
A13.       -2   : result = result + 60;
      // Assignment to a constant
A14.       -1   : result = result + 20;
      // Assignment to a Param.
A15.       1..k : result = result + 40 + check(ti,tp,v1..k)
      // Assignment to a context Var. recall check function to
      // trace back all the dependencies propagated at this context Var.
A16.      end;
A17.      found = true;
A18.     end;
A19.  until P = first_transition or found;
A20. end;
A21. if found then return result
A22. else return result + 60;
      // the dependency is left open ended
A23. end.
```

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

penalties and see this as an important area of future research.

3.3. The GA encoding

In this study, we use the encoding method presented in [11] where TPs are encoded as a sequence of integers. For a given EFSM with k states, the encoding method comprises three steps. First, calculate the numbers n_1, n_2, .., n_k of transitions that leave each state. Then, find the lowest common multiplier LCM of n_1, n_2, .., n_k. Finally, calculate the transition ranges r_1, r_2, .., r_k associated with each state where $r_i = LCM / n_i$. Thus, when considering a state s_i a number x in the range 1 to LCM uniquely identifies a transition: we simply divide x by r_i and round down to give the transition number. Thus, a sequence of numbers in the range 1 to LCM uniquely identifies a path through the EFSM.

Example 3. The considered EFSM has $k = 6$ states, $n_1 = 2$, $n_2 = 3$, $n_3 = 2$, $n_4 = 11$, $n_5 = 2$ and $n_6 = 1$. Thus the $LCM = 66$. The range associated with the first state is $r_1 = 33$ since $n_1=2$. Therefore [1..33] represents t_0 and [34..66] represents t_1. The range for the second state is $r_2 = 22$ since $n_2=3$. Thus [1..22] represents t_2, [23..44] represents t_3 and [45..66] represents t_4. Similarly we can compute all the transition ranges for other states.

4. Experiment and approach validation

In designing our experiment, we aimed to evaluate the efficiency of the proposed fitness metric in guiding a GA search for TPs that are feasible and easy to trigger using search. In order to achieve this, there are three factors to be considered. The first is related to the length of the TPs to be generated. Naturally, a short TP is likely to be easier to trigger since it has fewer transitions and hence fewer dependencies exist among its transitions. We therefore want to generate TPs that are relatively long. Since the EFSM machine at hand consists of 21 transitions, we consider TPs with length $n = 11$ to be long enough to avoid the impact of this factor on our results. The second factor is the number of input parameters required to trigger a given FTP. Since an FTP that requires fewer input parameters is typically easier to trigger, we have to generate FTPs that require relatively many input parameters. We set the number of the required input parameters to be $P_n = 10$. Both n and P_n were selected for illustration purpose and different values can be used to derive FTPs with different characteristics. The third factor is to determine how easy it is to trigger a generated FTP using search; we used a random test data generator to assess this. If we can quickly randomly find 10 suitable

input parameters to trigger a given FTP then the proposed FTP is considered to be easy to trigger.

The GA search that implemented the fitness metric was applied to the class 2 transport protocol EFSM in order to generate a set of 21 FTPs (n =11 and P_n = 10) that provides a transition coverage test suite (see Table 4). During TP generation, each path is first checked for the existence of a particular transition that this TP is intended to cover and for the value of P_n before the GA evaluates the path's fitness . Any TP that violates these constraints is considered invalid and given fitness 2000. Also, we randomly generated two alternative sets of 21 TPs (n =11 and P_n = 10) for the purpose of comparison (see Tables 5 and 6). Since we have a set of TPs rather than one single TP, a reset is applied to reinitialize the machine every time the random test data generator tries to trigger a given TP.

Both the GA and the random technique were implemented using the publicly available Genetic and Evolutionary Algorithm Toolbox (GEATbx) [31]. 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 tool website [31] and we record the values used here for the purpose of experiment replication.

An integer valued encoding with the value range of [1..66] was used for the GA. The population size was 25 individuals where each individual consisted of 11 variables to represent 11 transitions. The selection method was linear-ranking with a selective pressure set to 1.8. Discrete recombination was used whereas mutate integer mutation was applied. A random test data generator was implemented by setting the recombination and mutation methods to 'recnone' and 'mutrandint' respectively. An integer valued encoding was used to represent 10 input parameters with the range of values allowed being [0..1000].

Both techniques (GA and random) were allowed 1000 generations before search was terminated. Finally, we repeated the search with each technique 10 times for each subject TP.

4.1. Experimental results

A set of 21 FTPs generated using a GA which implemented the proposed fitness metric is reported in Table 4. Also, the two alternative sets of 21 TPs that were generated randomly for the purpose of comparison are reported in Tables 5 and 6. Since the complete set of results cannot fit in this paper, each table reports the transition sequence of each path together with its associated fitness, the average number of generations required by random test data generator in ten tries to trigger this path and whether or not this path was taken. From Table 4, we can state that all the 21 FTPs were feasible and easy to trigger. The random test data generator required, in most cases, only one generation to trigger these FTPs. However, some FTPs were associated with larger fitness values and required more attempts to be triggered. This suggests that FTPs with high fitness are hard to trigger using search.

Table 4. Results of the proposed approach

TPs generated by a GA approach	Fitness	Gen. (Avg.)	Taken
$\mathbf{t_0}$,t_2,t_{17},t_{18},t_1,t_6,t_{18},t_1,t_5,t_{10},t_{10}	8	1	yes
t_0,t_2,t_{17},t_{18},$\mathbf{t_1}$,t_6,t_{18},t_1,t_5,t_{10},t_{10}	8	1	yes
t_0,t_3,t_{19},t_1,t_6,t_{18},t_1,t_6,t_{18},t_0,$\mathbf{t_2}$	10	1	yes
t_1,t_5,t_{10},t_{17},t_{18},t_0,t_4,t_0,$\mathbf{t_3}$,t_{19},t_1	10	1	yes
t_0,$\mathbf{t_4}$,t_0,t_3,t_{19},t_1,t_6,t_{18},t_0,t_2,t_{10}	10	1	yes
t_1,t_6,t_{18},t_1,t_6,t_{18},t_1,$\mathbf{t_5}$,t_{10},t_{10},t_{10}	4	1	yes
t_1,$\mathbf{t_6}$,t_{18},t_0,t_2,t_{10},t_{17},t_{18},t_1,t_5,t_{10}	8	1	yes
t_0,t_2,t_{10},t_{16},t_{19},t_0,t_3,t_{20},t_1,t_5,$\mathbf{t_7}$	38	50	yes
t_1,t_6,t_{18},t_0,t_2,t_{10},t_{10},t_{10},$\mathbf{t_8}$,t_{10},t_{17}	40	200	yes
t_1,t_6,t_{18},t_0,t_2,t_{10},t_{10},$\mathbf{t_9}$,t_{16},t_{19},t_1	10	1	yes
t_1,t_6,t_{18},t_1,t_6,t_{18},t_0,t_2,$\mathbf{t_{10}}$,t_{10},t_{10}	4	1	yes
t_1,t_6,t_{18},t_0,t_4,t_1,t_6,t_{18},t_0,t_2,$\mathbf{t_{11}}$	40	175	yes
t_0,t_4,t_1,t_6,t_{18},t_1,t_6,t_{18},t_1,t_5,$\mathbf{t_{12}}$	34	1	yes
t_1,t_6,t_{18},t_0,t_4,t_1,t_5,$\mathbf{t_{13}}$,t_{17},t_{18},t_1	46	200	yes
t_1,t_6,t_{18},t_0,t_2,t_{10},$\mathbf{t_{14}}$,t_{16},t_{19},t_1,t_6	40	100	yes
t_0,t_3,t_{20},t_1,t_6,t_{18},t_1,t_5,$\mathbf{t_{15}}$,t_{10},t_{10}	34	1	yes
t_1,t_6,t_{18},t_1,t_5,t_{10},t_{10},t_{10},$\mathbf{t_{16}}$,t_{19},t_1	4	1	yes
t_1,t_6,t_{18},t_0,t_2,$\mathbf{t_{17}}$,t_{18},t_1,t_5,t_{10},t_{10}	8	1	yes
t_1,t_6,$\mathbf{t_{18}}$,t_1,t_5,t_{10},t_{10},t_{16},t_{19},t_1,t_5	8	1	yes
t_1,t_6,t_{18},t_1,t_5,t_{16},$\mathbf{t_{19}}$,t_0,t_3,t_{20},t_1	10	1	yes
t_1,t_5,t_{16},$\mathbf{t_{20}}$,t_1,t_5,t_{10},t_{10},t_{16},t_{19},t_1	8	1	yes

From Tables 5 and 6 we can observe that the fitness associated with each generated TP was larger than that of the FTPs produced by the GA search. Furthermore, the random test data generator failed, in most cases, to find input parameter values to trigger the TPs. This does not necessarily mean that these TPs are infeasible; however, TPs with a fitness value greater than 500 are very likely to be infeasible or hard to trigger since many of the TPs with fitness over 500 contained an instance of the infeasible TP case (which leads to 500 being added to the fitness value). For those TPs with fitness values less than 500 and not triggered randomly, we can state that they are complex enough not to be easily triggered by a random test data generator. Moreover, we observe that the successfully triggered TPs from both alternative sets have the lowest fitness values among TPs in these two sets. They also have larger fitness values than the FTPs generated by the GA search and required more generations to be triggered. This provides evidence that TPs with larger fitness values are more difficult to trigger. Finally, all TPs with a fitness value at most 34 required only one try by the random test data generator in order to be triggered.

Table 5. Results of the first alternative set

TPs generated randomly	Fitness	Gen. (Avg.)	Taken
$t_1,t_6,t_{18},\mathbf{t_0},t_2,t_7,t_7,t_{11},t_{15},t_{12},t_{10}$	204	1000	no
$t_0,t_2,t_8,t_7,t_{17},t_{18},\mathbf{t_1},t_5,t_{11},t_8,t_{16}$	1116	1000	no
$t_1,t_6,t_{18},t_0,t_4,t_0,\mathbf{t_2},t_9,t_8,t_{12},t_7$	588	1000	no
$t_0,\mathbf{t_3},t_{19},t_1,t_6,t_{18},t_0,t_4,t_1,t_5,t_8$	534	1000	no
$t_0,t_2,t_{15},t_{16},t_{20},t_0,\mathbf{t_4},t_0,t_2,t_8,t_{11}$	592	1000	no
$t_0,t_4,t_1,t_6,t_{18}t_0t_4,t_1,\mathbf{t_5},t_{12},t_8$	558	1000	no
$t_1,\mathbf{t_6},t_{18},t_1,t_5,t_8,t_{10},t_{10},t_9,t_{10},t_{15}$	616	1000	no
$t_0,t_3,t_{19},t_0,t_2,t_{15},\mathbf{t_7},t_{13},t_7,t_{13},t_7$	162	1000	no
$t_0,t_4,t_1,t_6,t_{18},t_1,t_5,t_{12},t_9,\mathbf{t_8},t_7$	588	1000	no
$t_1,t_5,t_{11},t_{15},t_7,t_9,t_9,t_{11},t_7,t_7,\mathbf{t_9}$	262	1000	no
$t_0,t_4,t_1,t_5,\mathbf{t_{10}},t_{11},t_8,t_{10},t_8,t_{15},t_{15}$	280	1000	no
$t_0,t_2,t_8,t_7,t_7,t_{12},t_{15},\mathbf{t_{11}},t_{10},t_{15},t_9$	940	1000	no
$t_0,t_3,t_{20},t_0,t_2,t_7,\mathbf{t_{12}},t_{11},t_7,t_{15},t_{16}$	210	1000	no
$t_1,t_5,t_7,t_9,t_7,t_9,t_{10},t_7,\mathbf{t_{13}},t_{15},t_{14}$	372	1000	no
$t_0,t_2,t_{12},t_7,\mathbf{t_{14}},t_8,t_7,t_7,t_{16},t_{20},t_1$	862	1000	no
$t_1,t_5,t_{14},t_9,t_7,t_8,\mathbf{t_{15}},t_{16},t_{19},t_0,t_2$	706	1000	no
$t_1,t_5,t_{13},t_{15},t_{10},t_{10},\mathbf{t_{16}}t_{19}t_1,t_5,t_{16}$	74	287	yes
$t_1,t_5,t_8,t_9,t_7,t_7,t_{11},t_{11},t_{15},t_9,\mathbf{t_{17}}$	860	1000	no
$t_1,t_6,\mathbf{t_{18}},t_0,t_4,t_1,t_5,t_{12},t_7,t_7,t_{13}$	194	1000	no
$t_1,t_6,t_{18},t_0,t_3,\mathbf{t_{19}},t_0,t_2,t_{15},t_{12},t_7$	88	350	yes
$t_1,t_6,t_{18},t_0,t_2,t_{16},\mathbf{t_{20}},t_1,t_5,t_7,t_{12}$	52	300	yes

Table 6. Results of the second alternative set

TPs generated randomly	Fitness	Gen. (Avg.)	Taken
$\mathbf{t_0},t_4,t_0,t_3,t_{20},t_0,t_4,t_1,t_5,t_{12},t_7$	64	312	yes
$t_0,t_2,t_{17},t_{18},\mathbf{t_1},t_5,t_{16},t_{20},t_1,t_5,t_8$	536	1000	no
$t_0,\mathbf{t_2},t_7,t_{10},t_{13},t_{15},t_{15},t_{11},t_8,t_{10},t_{17}$	164	1000	no
$t_0,\mathbf{t_3},t_{20},t_0,t_2,t_8,t_7,t_{14},t_{17},t_{18},t_0$	582	1000	no
$t_1,t_5,t_9,t_{17},t_{18},t_0,\mathbf{t_4},t_1,t_5,t_{11},t_{17}$	50	217	yes
$t_0,t_3,t_{19},t_1,\mathbf{t_5},t_7,t_{16},t_{20},t_0,t_2,t_8$	562	1000	no
$t_1,t_5,t_{12},t_8,t_{15},t_{17},t_{18},t_1,\mathbf{t_6},t_{18},t_1$	582	1000	no
$t_0,t_2,t_{12},t_9,t_{15},t_{11},t_8,\mathbf{t_7},t_{10},t_{17},t_{18}$	648	1000	no
$t_1,t_5,t_{12},\mathbf{t_8},t_{16},t_{20},t_0,t_3,t_{20},t_0,t_4$	564	1000	no
$t_0,t_3,t_{19},t_0,t_2,t_7,t_{15},t_{13},\mathbf{t_9},t_8,t_{17}$	700	1000	no
$t_0,t_2,\mathbf{t_{10}},t_{12},t_7,t_9,t_{16},t_{20},t_0,t_3,t_{19}$	70	243	yes
$t_1,t_5,t_7,t_{13},t_{10},\mathbf{t_{11}},t_8,t_{17},t_{18},t_0,t_4$	116	1000	no
$t_1,t_5,t_{17},t_{18},t_0,t_2,t_9,\mathbf{t_{12}},t_8,t_{17},t_{18}$	568	1000	no
$t_0,t_4,t_0,t_3,t_{20},t_0,t_2,t_7,\mathbf{t_{13}},t_{15},t_8$	610	1000	no
$t_1,t_5,t_7,t_7,t_{12},t_9,\mathbf{t_{14}},t_{10},t_{17},t_{18},t_0$	186	1000	no
$t_0,t_2,t_7,t_9,t_{11},t_{10},t_7,t_{13},\mathbf{t_{15}},t_9,t_{17}$	140	1000	no
$t_1,t_5,t_{13},t_{14},t_7,t_{10},t_8,\mathbf{t_{16}},t_{19},t_0,t_4$	142	1000	no
$t_1,t_5,t_8,t_{16},t_{20},t_1,t_5,\mathbf{t_{17}},t_{18},t_1,t_5$	536	1000	no
$t_1,t_6,\mathbf{t_{18}},t_1,t_5,t_8,t_{10},t_{15},t_8,t_{12},t_7$	762	1000	no
$t_0,t_3,\mathbf{t_{19}},t_0,t_4,t_1,t_5,t_{12},t_8,t_{16},t_{20}$	564	1000	no
$t_0,t_2,t_8,t_8,t_9,t_{10},t_{17},t_{18},t_0,t_3,\mathbf{t_{20}}$	966	1000	no

4.2. Threats to validity

Threats to external validity are the conditions that restrict our ability to generalize our results. In this study, this can be related to the case study used. Although the case study is nontrivial, using other case studies would increase the confidence in the results. This threat has been limited by using an EFSM model used to evaluate other test techniques. Another factor is the penalty values we used: although these led to significant results, it might be possible to adjust the values further to produce a fitness metric with improved accuracy. Furthermore, the length of the generated FTPs was relatively short and for longer paths, the length could also contribute to a path fitness value. These require further research with additional EFSM case studies.

5. Conclusion

Testing from EFSM models is usually complicated by the presence of infeasible paths and generating a set of TPs that are feasible is a challenging task. In this paper, we present a fitness metric based on the analysis of interdependencies among guards and actions found in a given TP in order to estimate the traversal complexity of this TP and hence its feasibility. The fitness metric is simple to compute and so could be incorporated into a search based testing technique with the aim of producing feasible paths that satisfy the test criterion and are relatively easy to trigger using search.

In the experiment, the proposed fitness metric was utilized to guide a GA to find a set of FTPs that achieves the transition coverage adequacy criterion. Also, we randomly generated two alternative sets of TPs for comparison. We observed that the proposed fitness metric successfully guided the GA to produce a set of FTPs that have low fitness values and can be easily triggered. The two alternative sets of TPs had higher fitness values and the random test data generator failed, in most cases, to trigger them. The results of the experiment suggest that the proposed fitness metric can effectively estimate the traversal complexity of a given TP and so the likelihood of this TP being feasible.

Further research will apply the approach to additional EFSM case studies and investigate how different penalty values in the fitness metric affect its ability to guide the search towards FTPs that are relatively easy to trigger. It may also be possible to incorporate additional information into the fitness metric in order to make it more precise. We will investigate multi-objective search technique that utilizes the fitness metric to search for paths that satisfy a standard test criterion and are easy to trigger. There is also a need for experiments to explore the effect on fault detection of using paths that are easy to trigger.

6. References

[1] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on*, vol. 16, pp. 870-879, 1990.

[2] P. McMinn, "Search-based software test data generation: a survey: Research Articles," *Software Testing, Verification & Reliability*, vol. 14, pp. 105-156, 2004.
[3] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating software test data by evolution," *Software Engineering, IEEE Transactions on*, vol. 27, pp. 1085-1110, 2001.
[4] A. Petrenko, S. Boroday, and R. Groz, "Confirming configurations in EFSM testing," *Software Engineering, IEEE Transactions on*, vol. 30, pp. 29-42, 2004.
[5] R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software*, vol. 62, pp. 21-46, 2002.
[6] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, pp. 1090-1123, 1996.
[7] A. Y. Duale and M. U. Uyar, "A method enabling feasible conformance test sequence generation for EFSM models," *Computers, IEEE Transactions on*, vol. 53, pp. 614-627, 2004.
[8] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," presented at COMPSAC 2001. 25th Annual International, pp. 489-495, 2001.
[9] S. T. Chanson and J. Zhu, "A unified approach to protocol test sequence generation," presented at Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE, pp. 106-114, 1993.
[10] H. Ural and B. Yang, "A test sequence selection method for protocol testing," *Communications, IEEE Transactions on*, vol. 39, pp. 514-523, 1991.
[11] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo, "Generating feasible input sequences for extended finite state machines (EFSMs) using genetic algorithms," in *Proceedings of the 2005 conference on Genetic and evolutionary computation*. Washington DC, USA: ACM, pp. 1081-1082, 2005.
[12] A. Y. Duale, M. U. Uyar, B. D. McClure, and S. Chamberlain, "Conformance testing: towards refining VHDL specifications," presented at Military Communications Conference Proceedings, 1999. MILCOM 1999. IEEE, pp. 140-144, 1999.
[13] R. M. Hierons, T.-H. Kim, and H. Ural, "On the testability of SDL specifications," *Computer Networks*, vol. 44, pp. 681-700, 2004.
[14] L.-S. Koh and M. T. Liu, "Test path selection based on effective domains," presented at Network Protocols, 1994. Proceedings., 1994 International Conference on, Boston, MA, pp. 64-71, 1994.
[15] T. Ramalingom, K. Thulasiraman, and A. Das, "Context independent unique state identification sequences for testing communication protocols modelled as extended finite state machines," *Computer Communications*, vol. 26, pp. 1622-1633, 2003.
[16] B. Sarikaya, G. v. Bochmann, and E. Cerny, "A Test Design Methodology for Protocol Testing," *Software Engineering, IEEE Transactions on*, vol. SE-13, pp. 518-531, 1987.
[17] C.-J. Wang and M. T. Liu, "Generating test cases for EFSM with given fault models," presented at Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future. IEEE, pp. 774-781, 1993.
[18] C.-H. Shih, J.-D. Huang, and J.-Y. Jou, "Stimulus generation for interface protocol verification using the nondeterministic extended finite state machine model," presented at High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International, pp. 87-93, 2005.
[19] G. V. Bochmann, "Specifications of a simplified transport protocol using different formal description techniques," *Computer Networks and ISDN Systems*, vol. 18, pp. 335-377, 1990.
[20] J. H. Holland, *Adaptation in natural and artificial systems*. Cambridge, MA: MIT Press, 1992.
[21] G. E. Liepins and M. R. Hilliard, "Genetic algorithms: Foundations and applications," *Annals of Operations Research*, vol. 21, pp. 31-57, 1989.
[22] X. Yao, "Global optimisation by evolutionary algorithms," presented at Parallel Algorithms/Architecture Synthesis, 1997. Proceedings. Second Aizu International Symposium, pp. 282-291, 1997.
[23] M. Srinivas and L. M. Patnaik, "Genetic algorithms: a survey," *Computer*, vol. 27, pp. 17-26, 1994.
[24] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. Boston, Massachusetts, USA: ACM, pp. 108-118, 2004.
[25] K-C. Tai, "A program complexity metric based on data flow information in control graphs," in *Proceedings of the 7th international conference on Software engineering*. Orlando, Florida, United States: IEEE Press, pp. 239-248, 1984.
[26] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. San Diego, California, United States: IEEE Press, pp. 439-449, 1981.
[27] K.-T. Cheng and A. S. Krishnakumar, "Automatic generation of functional vectors using the extended finite state machine model," *ACM Transactions on Design Automation of Electronic Systems.*, vol. 1, pp. 57-79, 1996.
[28] T. A. Dahbura, K. K. Sabnani, and M. U. Uyar, "Formal methods for generating protocol conformance test sequences," *Proceedings of the IEEE*, vol. 78, pp. 1317-1326, 1990.
[29] A. Petrenko, G. v. Bochmann, and M. Yao, "On fault coverage of tests for finite state specifications," *Computer Networks and ISDN Systems*, vol. 29, pp. 81-106, 1996.
[30] R. M. Hierons and M. Harman, "Testing conformance of a deterministic implementation against a non-deterministic stream X-machine," *Theoretical Computer Science*, vol. 323, pp. 191-233, 2004.
[31] H. Pohlheim, "GEATbx - Genetic and Evolutionary Algorithm Toolbox," http://www.geatbx.com, 2008.