Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 309 (2014) 13–29

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

An Approach to Generating Test Data for EFSM Paths Considering Condition Coverage

Gongzheng Lu [1,2]

School of Computer Engineering and Science
Shanghai University
Shanghai 200072, China

School of Computer Engineering
Suzhou Vocational University
Suzhou 215104, China

Huaikou Miao [3]

School of Computer Engineering and Science
Shanghai University
Shanghai 200072, China

Shanghai Key Laboratory of Computer Software Testing & Evaluating
Shanghai 201114, China

Abstract

Model-based test case generation has become a hotspot, and automatic generation of test data is difficult in this area. In this paper, system model is represented by extended finite state machine(EFSM), and genetic algorithm is used to generate test data for EFSM paths. When computing the fitness of an individual, the branch distance and the ratio of uncovered conditions of the individual are considered. In experiments, the proposed method is compared with the Kalaji's, and the results show that our method has a better effect and can get higher quality test data.

Keywords: EFSM, Test Data, Genetic Algorithm, Condition Coverage

1 Introduction

Software testing is an effective software quality assurance technology. In software testing, we first design test cases which consist of a finite sequence of inputs and outputs for the system under test, then we execute each test case on the system under test and observe the execution result. At last we compare the execution result

[1] This work is supported by the National Natural Science Foundation of China (NSFC) under grant No.61073050 and No.61170044.

[2] Email: lugz@shu.edu.cn

[3] Email: hkmiao@shu.edu.cn

http://dx.doi.org/10.1016/j.entcs.2014.12.003

with the expected result claimed in the test case. If the execution result is equal to the expected result, we say the test case is pass else is fail. Testing can constitute up to 50% of the overall software development cost. Therefore, we must develop methods to reduce the cost associated with the testing process. Since, manual testing is low efficient, error prone and non reproduction and increases the software testing cost. Automated testing is an inevitable trend, and now model-based testing is a research hot.

In model-based testing, it is necessary to use a formal model to specify the system specification, and to generate test cases according to a given test criterion. Two kinds of models that can be used to specify the system under test are finite state machines(FSMs) [1] and extended finite state machines(EFSMs) [2]. The FSM is consisted of a finite set of states and transitions. A transition in FSM includes four parts: entering state, exiting state, input and output. FSM can only model the control part of the system. EFSM extends the FSM with variables. The transitions in EFSM can also have predicate(guard condition) and operation(assignment), so the EFSM can specify both the control part and the data part of the system. Because EFSM can specify the data part of the system and the aim of this paper is to generate test data for the test cases get from the formal model, we choose EFSM to model the system under test.

The main coverage criteria about FSM and EFSM are state coverage, transition coverage and transition pair coverage. State coverage requires that all states of the EFSM should be exercised by the test cases. Transition coverage requires that all the transitions of the EFSM are executed by the test cases. And transition pair coverage requires that all the adjacent transition pairs should be executed by the test cases. Transition coverage criterion is stronger than state coverage criterion, that is, the test cases satisfying the transition coverage criterion must satisfy the state coverage criterion. And transition pair coverage criterion is stronger than transition coverage criterion, that is, the test cases exercising all the transition pairs of the system are sure to exercise all the transitions.

In EFSM, each test case defines a transition path(TP) which comprised of a sequence of transitions through the EFSM. For example, transition coverage requires that each transition of the EFSM is executed by at least a TP corresponding to a test case. So, when generating test cases from EFSM model, we should firstly generate a set of TPs that satisfy the given coverage criterion and then generate test cases which can trigger these TPs. However, the transition may contain guard condition and operation, the TP generated from the EFSM may be infeasible. The feasibility of the TPs is out of the scope of this paper. We only generate test data for these feasible TPs. Even if a path is feasible, it is still difficult to find test data to execute it. Because the range of the input variables in the TPs is usually large, but the set of values of the variables satisfied with the guard conditions may be small.

There are two types of methods for generating test data: constraint-based testing(CBT) [3] and search-based testing(SBT) [4]. Constraint-based method uses symbolic execution to get path constraints, which only relate with input parameters. And it then solves each path constraint using a constraint solver and gets the

test data satisfying the constraint. Search-based method is similar to constraint-based method, it also needs to get the path constraint firstly. The difference is that search-based testing converts the test data generation into function optimization, and the individual with best fitness(fitness is 0) got by search algorithm is as test data. This function optimization can be solved by metaheuristic techniques, like genetic algorithm, ant colony and particle swarm optimization.

In this paper, extended finite state machine(EFSM) is used to model the system under test, and genetic algorithm is used to generate test data for EFSM paths. We analyze the fitness function proposed by Kalaji [5], use a small example to explain the shortcomings of Kalaji's and Lefticaru's method [15] which is similar to the method proposed by Kalaji. And in order to solve the problems existing in Kalaji's method, we propose a new fitness function for generating test data for EFSM paths. This new fitness function combines the branch distance with ratio of uncovered conditions to compute the fitness of an individual. It improves Kalaji's function, in terms of average number of generation, success rate and the average violation ratio of conditions of test data. In the experiments, we use three EFSM models to compare our method with the Kalaji's, and the results show that our method has a better effect and can get higher quality test data.

The rest of the paper is organized as follows: the related works are presented in section 2, and the background is indicated in section 3, the method we proposed is provided in section 4, section 5 are the experiments and the results analysis, and section 6 is the concluding section.

2 Related Works

There have been a lot of research on test data generation for program before [6–8], and many researchers have been applied the search-based method into this field [9–12]. But there has been a relatively little work that generates test data from EFSM model. J.Zhang et al. [13] proposed a path-oriented test data generation method, they first derived the path constraint using symbolic execution and then solved the path constraint using a constraint solver, at last the test data is obtained. The limitation of this method is that it cannot deal with nonlinear constraints. R.Lefticaru et al. [14] given a fitness evaluation method to get input sequence for paths, the fitness function proposed by Tracey was applied to each transition of the path, and the fitness of the path was defined by viewing each transition in the path as a critical node. Its limitation is that it requires that each transition cannot include inner path, else the Tracy's fitness cannot used in his method. A.S. Kalaji et al. [5] used genetic algorithm to generate test data for EFSM, and the fitness function is consisted of branch distance and approach level. This fitness function may drop test data with better fitness and can not give the same selection probability to the test data which have the same distance away from the interval of range. R.Lefticaru et al. [15] also improved the Kalaji's fitness function, they decomposed the path into independent sub-paths and computed the fitness of each sub-path according to Kalaji's method, and the global fitness of the path is the sum

of the fitness of each sub-path. Their method rewards the individuals that satisfy more constraints of the path which similar to our method. The shortcomings of their method is that it still meets the problems occurring in Kalaji's method and we can not always find the independent sub-paths for each path. Our experiment shows that the effect of the fitness function proposed by us is better than Kalaji's.

3 Preliminaries

3.1 Extended Finite State Machine

A finite state machine consists of a finite set of states, transitions, inputs and outputs. Because FSM can't specify the data part of the system, we use extended finite state machine to model the system under test. The EFSM extends the FSM with context variables, predicates and operations.

An EFSM is a six-tuple (S, s_0, V, I, O, T), where S is a finite set of states, s_0 is an initial state, V is a finite set of context variables, I is a finite set of inputs, O is a finite set of outputs, and T is a finite set of transitions. The transition $t \in T$ is a five-tuple (s_s, i, g, op, s_e), where s_s is the source state of t, $i \in I$ is the input which may associated with input parameters, g is a logic expression called guard condition, op is an operation consisted of assignment statements or output statements, and s_e is the target state of t. If the state is s_s, the input is i and the guard condition is satisfied, then transition $t{=}(s_s, i, g, op, s_e)$ will be triggered, meanwhile the operation in op is executed and the EFSM converts to state s_e. g and op both can contain input parameters and context variables. Here, we only consider deterministic EFSMs. An EFSM is deterministic if for the transitions with same inputs that leave a state, just the guard condition of one transition is satisfied at one time [16]. The definitions and details about EFSM can be found in [2].

3.2 Symbolic Execution

Symbolic execution [17] is a method to analyze the program statically. It uses symbolic values instead of real values to execute program, and the result is expressions over symbols. It is useful to understand the relationship between inputs and outputs.

When symbolic execution is used for test data generation, the problem of test data generation can be translated into the problem of solving the symbolic expressions that result from executing the path using symbolic values. For example, given a transition path $t_1t_2t_3$, the predicate on each transition is $x > 0$, $y < 15$ and $z \geq 10$ separately. We substitute symbols a, b and c for variables x, y and z respectively. After executing the path using symbolic values, we obtain a symbolic expression $a > 0\ AND\ b < 15\ AND\ c \geq 10$. The problem of generating test data for path $t_1t_2t_3$ now is converted into the problem of finding the solution of the symbolic expression $a > 0\ AND\ b < 15\ AND\ c \geq 10$.

3.3 Data Flow Dependence

Though there exists input variables and context variables in EFSM, the symbolic expression is usually expressed only by input variables. The context variables in the path must be substituted by the corresponding values and input variables using data flow dependence analysis.

Given a variable v, if it is an input parameter in transition t or is assigned in an operation of the transition t, then v is said to be defined in t, written as $def(t)$. If v is referenced in a predicate (p-use) or in the operation of the transition t, then v is said to be used in t, written as $use(t)$. Given a path between two transitions t_i and t_j, $v \in def(t_i)$ and $v \in use(t_j)$, if v is not defined after t_i and before t_j then the path from t_i to t_j is a definition clear path for v. (t_i, t_j) forms a definition-use pair for v, and there is data flow dependence between t_i and t_j [18].

After getting the data flow dependence for the context variables on each transition, these context variables are substituted by the values and input parameters on which they depend using backward substitution. Back substitution deals with the transitions from back to front in a path, substitutes the definitions the variables depending for the variables used in the transition, at last the path expression is the symbolic expression represented only by input variables.

3.4 Genetic Algorithms

Genetic algorithm(GA) is a kind of randomized method evolved from biological evolution laws. It has parallel and global optimization searching capability, can obtain and guide the optimized search space automatically and can adjust the direction of searching adaptively. These properties of genetic algorithm have been widely used in combinatorial optimization, machine learning and other fields. Nowadays, it has been applied into software testing, especially in search-based software testing.

Genetic Algorithms(GA) [19] are powerful metaheuristic technique. In the genetic algorithm, the candidate solution(also called individual) is called chromosome consisting of genes. The GA cycle consists of the main operators: evaluation, selection, crossover and mutation. The GA evaluates the fitness of each individual using the fitness function. After individuals are evaluated, a selection based on fitness is made. When the individuals are selected as parent individuals according to the selection policy, then the cross operator is applied to generate new individuals. The mutation operator is another way can be used to generate new individuals, which only acts on an individual at a time and changes the values of some of the individual's genes randomly.

Search-based software testing translates software testing problem into optimization problem. In search-based testing, in order to generate test data for EFSM paths, we need to compute the symbolic expression for each path using data flow dependence analysis, and then we use genetic algorithm to search the optimal solution for the symbolic expression which is viewed as a function to be solved. The test data correspond to the individuals. First, we need to encode the test data using the encoding mechanism provided by genetic algorithm. The encoding of the

Table 1
The fitness calculation proposed by Tracey et al.

Guard	Fitness calculation
$Boolean$	If $TRUE$ then 0 else k
$a == b$	If $abs(a-b) == 0$ then 0 else $abs(a-b)+k$
$a! = b$	If $abs(a-b)! = 0$ 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 $(b-a)+k$
a	Negation is moved inwards and propagated over a

candidate solutions have binary encoding, integer valued encoding and real valued encoding. In this paper, we use real valued encoding to represent the test data. We use an evaluation method called fitness function to measure how good each test data is. The fitness function assigns each test data a positive number, and this number estimates how far it is from being an acceptable solution can trigger the path. The optimization problem is usually a minimization one, the test data with lower fitness is better, and the test data with fitness zero is an acceptable solution. When the representation of the test data is decided and the fitness function is defined, we can use genetic algorithm to generate test data for EFSM paths.

4 The proposed approach

Kajaji's fitness function was an extension of Wegener's method [5]. Kalaji's fitness function is consisted of branch distance and approach level. Branch distance is a fitness evaluation method which is contained by Tracey et al, the detail is in table 1. For example, for condition $x > 0$, if $(0-x) < 0$ then its branch distance is 0 which states that the current value of x satisfies the given condition. And if $(0-x) \leq 0$ then the branch distance is not zero(branch distance is $(0-x)+k$, $k > 0$ is a constant), it reflects how close the selected value was to achieving the condition.

Approach level was proposed by Wegener. It measures how close the test data was to executing the target statement. The distance is computed by subtracting one from the number of critical nodes away from the target. A critical node is a conditional statement at which the execution flow may divert. And this method can only be used to cope with a single target at one time. In order to cope with a sequence of transitions, Kalaji improved Wegener's method, defined the approach level of a transition which is computed by subtracting one from the number of critical transitions away from the target transition, and manipulated a path which contains several transitions in the way similar to nested IF statement. Kalaji's

fitness function is given as follows:

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

$$function\_distance = norm(branch\_distance) + transition\_approach\_level \quad (2)$$

$$transition\_approach\_level = NumOfCriticalTransAwayFromTarget - 1 \quad (3)$$

$$path\_fitness = norm(function\_distance) \quad (4)$$

$norm()$ in equation (1) is a normalization function. The branch distance is normalized to a value in the range [0...1]. The fitness of a path can be derived in a similar manner as the method proposed by Wegener for nested predicates. Given a path, the function distance is computed for each transition that has guard condition by applying Wegener's method(Eq.(2)). Then, any transition that has guard condition is considered a critical transition and so the value of *transition_approach_level* is derived by subtracting 1 from the number of critical transitions away from the target transition(Eq.(3)). Finally, the path fitness is the sum of the *transition_approach_level* and the normalized value of *branch_distance*.

In EFSM, the guard condition of a transition can be connected by logic operators AND and OR. The guard condition connected by AND operator can be represented as nested IF statements. If the guard condition connected by OR operator, a transition can be splited into a number of transitions equal to the number of OR operators+1, and we can calculate the fitness of each transition separately, and the fitness of the guard condition is the minimum fitness among the transitions.

We take Kalaji's fitness function, for example, to compute the fitness of a path which only has one transition t_1, and the predicate on t_1 is $0 \leq x$ AND $x \leq 15$.

if $0 \leq x$
  if $x \leq 15$
    $result = 0$ //candidate solution is an acceptable solution
  else $result = norm(x - 15)$
  end
else $result = norm(0 - x) + 1$
end

We assume that there are two inputs $x = -1$ and $x = 16$. According to Kalaji's fitness function, when $x = -1$, the condition $0 \leq x$ is not satisfied, so the fitness of the predicate is $result = norm(1)+1$. When $x = 16$, the condition $0 \leq x$ is satisfied but $x \leq 15$ is not satisfied, so the fitness of the predicate is $result = norm(1)$. In fact, the distance -1 and 16 away from the interval [0,15] are both 1, they should have the same fitness and have the same probability to be selected in genetic algorithm. When evaluating their fitness by Kalaji's method, the fitness are not equal. $x = 16$ has a lower fitness, so it may be selected to generate new individual.

Then we calculate the fitness of the path using Lefticaru's method when the inputs are $x = -1$ and $x = 16$. We can get the same result as Kalaji's method.

Next, Let us consider another situation. There are two transitions t_1 and t_2 in a path, the predicate on t_1 is $x1 \geq 16$ AND $x2 \leq 9$, and the predicate on t_2 is $x3 \geq 7$ AND $x4 \leq 0$, the fitness of the path computed using Kalaji's method is as follows:

if $x1 \geq 16$

if $x2 \leq 9$
  if $x3 \geq 7$
    if $x4 \leq 0$
      $result = 0$
    else $result = norm(x4 - 0)$
    end
  else $result = norm(7 - x3) + 1$
  end
else $result = norm(x2 - 9) + 2$
end
else $result = norm(16 - x1) + 3$
end

We assume that there are two inputs $(x1, x2, x3, x4)$=$(15, 8, 8, -1)$ and $(x1, x2, x3, x4) = (17, 10, 6, 1)$. The first input satisfies all conditions except the first one, and the second input only satisfies the first condition. The fitness of the first input is $result = norm(1)+3$, and the fitness of the second input is $result = norm(1)+2$, so the probability of the second input to be selected as parent individual is higher than the first input. Only the value of variable $x1$ is not satisfied in the first input, and all the value of the variables except $x1$ are not satisfied in the second input, so we can get the acceptable solution from the first input easier than from the second one in fact.

We also compute the fitness of the path using Lefticaru's method when the inputs are $(x1, x2, x3, x4)$=$(15, 8, 8, -1)$ and $(x1, x2, x3, x4) = (17, 10, 6, 1)$. We can see $t1$ and $t2$ are independent. So there are two independent sub-paths, and the fitness of each sub-path can be calculated separately. The fitness of the first input is $result = norm(1) + 1$ and the fitness of the second input is $result = 2*norm(1)+1$. The probability of the first input to be selected as parent individual is higher than the second input, because the first input satisfies more conditions than the second one, it gets lower fitness. Although, its result conforms the facts, the fitness calculation of each independent sub-path uses nested IF forms similar to in Kakaji's method, Lefticaru's method still meets the problems occurring in Kalaji's method, and we cannot always find the independent paths in the path set, when there are no independent paths in the path set, the effect of Lefticaru's method is the same as Kalaji's. So we do not compare our method with Lefticaru's method in our experiments.

In order to solve the problems of the Kalaji's method mentioned above, we propose a fitness function considering the ratio of the conditions in the path covered by each individual. Our fitness function combines branch distance with the ratio of uncovered conditions. Assuming the number of the conditions appearing in the path is n, and the number of the conditions unsatisfied by an individual is $m(m \leq n)$, our fitness function can be represented as follows:

$$uncovered\_condition\_ratio = \frac{m}{n} \tag{5}$$

$$\begin{aligned} path\_fitness = &\sum_{i=1}^{m} norm(branch\_distance_i) \\ &+ uncovered\_condition\_ratio \end{aligned} \tag{6}$$

In our method, the guard conditions on the transitions in the path are not represented as nested IF statements but as parallelism form. And the guard condition linked by operator AND is also represented as parallelism form, not nested IF form. The guard condition connected by operator OR is splited into separate transitions, and the fitness of the guard condition is the minimum fitness among these transitions.

We use our proposed method to calculate the fitness of predicate $0 \leq x \; AND \; x \leq 15$ on transition t. This predicate has two conditions:

if $0 \leq x$
    $result1 = 0$
else $result1 = norm(0 - x) + 1/2$
end
if $x \leq 15$
    $result2 = 0$
else $result2 = norm(x - 15) + 1/2$
end
$result = result1 + result2$

Here, we still assume that there are two inputs $x = -1$ and $x = 16$. According to our fitness function, when $x = -1$, the condition $0 \leq x$ is not satisfied, $result1 = norm(1) + 1/2$, and the condition $x \leq 15$ is satisfied, $result2 = 0$, finally the fitness is $result = norm(1) + 1/2$. When $x = 16$, the condition $0 \leq x$ is satisfied, $result1 = 0$, and the condition $x \leq 15$ is not satisfied, $result2 = norm(1) + 1/2$, the fitness of the predicate is $result = norm(1) + 1/2$. So the two inputs have the same fitness. When two individuals have the same distance away from the bounds of the range, then our fitness function assigns them the same fitness, and they have the same probability to be selected.

We rewrite another example which is calculated by Kalaji's method above using our method. The predicate in this example has four conditions, and the fitness can be calculated as follows:

if $x1 \geq 16$
    $result1 = 0$
else $result1 = norm(16 - x1) + 1/4$
end
if $x2 \leq 9$
    $result2 = 0$
else $result2 = norm(x2 - 9) + 1/4$
end
if $x3 \geq 7$
    $result3 = 0$

else $result3 = norm(7 - x3) + 1/4$
end
if $x4 \leq 0$
$result4 = 0$
else $result4 = norm(x4 - 0) + 1/4$
end
$result = result1 + result2 + result3 + result4$

We still assume that there are two inputs $(x1, x2, x3, x4) = (15, 8, 8, -1)$ and $(x1, x2, x3, x4) = (17, 10, 6, 1)$. The fitness of the first input is $result = norm(1) + 1/4$, and the fitness of the second one is $result = 3 * norm(1) + 3/4$. So the first input has the higher probability selected as parent individual than the second one, which conforms the facts. When the individual satisfies more conditions, that is the uncovered ratio of the conditions is lower, then our fitness function assigns it a lower fitness, and it has a higher probability to be selected.

The procedure of our method is as follows:

1. Obtain the path constraint for each transition path in EFSM using symbolic execution and data flow dependency analysis. And the path constraint only contain input variables.

2. Represent the path constraint using parallelism IF statements. The path constraint is combined of the guard condition on each transition linked by AND operators. According to our method, these guard condition are represented as parallelism IF statements, and the conditions in each guard condition linked by AND operators are also represented as parallelism form, but each guard condition linked by OR operators are splited into separate transitions, and the fitness of the guard condition is the minimum fitness among these transitions.

3. Use genetic algorithm to generate test data for each EFSM path. In the genetic algorithm, we evaluate the fitness of the test data using the fitness function proposed in this section.

We use an example to explain our method. $t.g$ and $t.op$ represent the guard condition and operation of transition t respectively. Assume there is a path $t_1t_2t_3t_4$. $t_1.g$ is $True$, $t_1.op$ is $v1 = pv1\ AND\ v2 = pv2\ AND\ v3 = pv3\ AND\ v4 = pv4$, $t_2.g$ is $v1 \geq 11\ AND\ v1 \leq 25\ AND\ v2 \geq 50\ AND\ v2 \leq 85$, $t3.g$ is $v3 \geq 11\ AND\ v3 \leq 25$, $t4.g$ is $v4 \geq 36\ OR\ v4 \leq 10$, and the operations of t_2, t_3 and t_4 are all NULL. According to data flow dependency analysis, t_2, t_3 andt_4 depend on t_1. The context variables occurring in the guard conditions of these transitions should be substituted by the definitions defined in the operation of t_1 and then the path constraint is $pv1 \geq 11\ AND\ pv1 \leq 25\ AND\ pv2 \geq 50\ AND\ pv2 \leq 85\ AND\ pv3 \geq 11\ AND\ pv3 \leq 25\ AND\ (pv4 \geq 36\ OR\ pv4 \leq 10)$. Because the guard condition linked by OR operations are divided into separate transitions, and the guard condition is satisfied when as long as there is one separate transition's condition is satisfied, the number of the conditions in the guard condition linked by OR operators is counted as 1. Then the number of the conditions of above path constraint is 7. In genetic algorithm, the fitness of the test data generated for path $t_1t_2t_3t_4$ can be calculated in following:

if $pv1 \geq 11$

$result1 = 0$
else $result1 = norm(11 - pv1) + 1/7$
end
if $pv1 \leq 25$
$result2 = 0$
else $result2 = norm(pv1 - 25) + 1/7$
end
if $pv2 \geq 50$
$result3 = 0$
else $result3 = norm(50 - pv2) + 1/7$
end
if $pv2 \leq 85$
$result4 = 0$
else $result4 = norm(pv2 - 85) + 1/7$
end
if $pv3 \geq 11$
$result5 = 0$
else $result5 = norm(11 - pv3) + 1/7$
end
if $pv3 \leq 25$
$result6 = 0$
else $result6 = norm(pv3 - 25) + 1/7$
end
if $pv4 \geq 36$
$result7 = 0$
else $result7 = norm(36 - pv4) + 1/7$
end
if $pv4 \leq 10$
$result8 = 0$
else $result8 = norm(pv4 - 10) + 1/7$
end
if $result7 \leq result8$
$result9 = result7$
else $result9 = result8$
end
$result = result1 + result2 + result3 + result4 + result5 + result6 + result9$

5 Experimental results

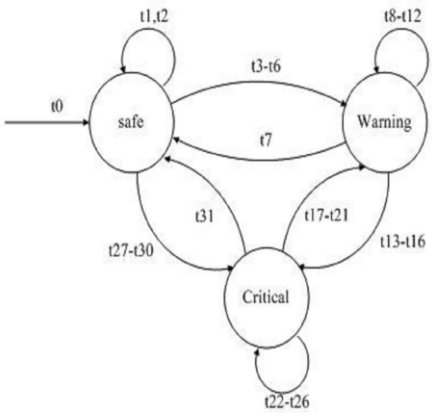
In this section, we generate test data for three EFSMs by using the approach we proposed, and compare the results with the test data generated by using Kalaji's method.

The experiments are carried out on three different EFSMs: In Flight Safety System EFSM, Transport Protocol EFSM and Lift System EFSM [5] in Figure 1.

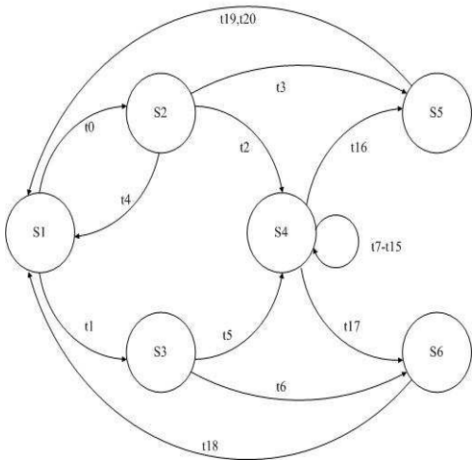
The detailed specification of the transitions of these EFSMs can be seen in literature [5]. In order to compare our method with Kalaji's, we analyze the results from three factors. The first factor is the average number of generations to generate test data satisfying all the conditions successfully. The second factor is the success rate of generating test data when the two methods sometimes can only generate test data that do not 100% satisfy all the conditions. When the method can generate test data which satisfy all the conditions in one run, we say the method is succeed in this run. And the last one is the average ratio of constraints violated by test data, when the two methods can only generate test data which do not satisfy all the conditions in all runs.

Our approach and Kalaji's are both implemented by using Genetic Algorithm and Direct Search Toolbox for Mathlab 7.0. The individual is represented by real valued encoding, stochastic uniform is selected as selection policy, scatter is used as crossover function, crossover probability is 0.8, the mutation function is the Gaussian function, the population size is set to 20, the initial range of each variable is [0...100]. The search will terminate when the fitness value 0 is achieved or the maximum number of 1000 generations is reached. We repeat the search ten times for each transition path, and we compute the average in the ten runs.

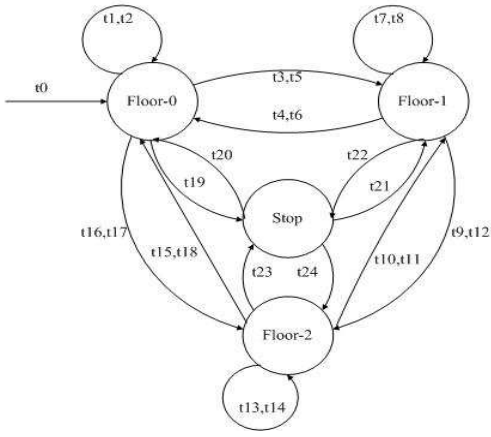
According to the transition criterion, we can generate 20 transition paths for In Flight Safety System EFSM. And the average number of generations to generate test data for it are shown in Figure 2(a). The average number of generations of the two methods to generate test data for these paths except $p2, p17, p18, p19, p20, p21$ are equal which is about 52. This is because the conditions of these paths are simple and only refer to a single variable. In this situation, the performance of these two methods is the same. When we generate test data for path $p2, p17, p18, p19, p20, p21$, the average number of generations of our method are greater than Kalaji's. Both methods can only generate test data that do not satisfy all the conditions for these paths sometimes in the ten runs, but the success rate of our method to generate test data is higher than Kalaji's. The success rate of the two methods to generate test data for the six paths can be seen in Figure 2(b). The reason that both methods can only generate test data that do not satisfy all the conditions for these paths sometimes is that the conditions in these paths are more complex and refer to more variables. Kalaji's method assigns an individual which does not satisfy the outer condition a higher fitness. And this individual may satisfy more inner conditions, and the acceptable solution may be generated from this individual, but it may be discarded by Kalaji's method, which result in a bad optimization fitness and can only generate test data that do not satisfy all the conditions for these paths with more complex conditions. Our method considers the coverage ratio of conditions of the individual, the higher coverage ratio the individual has the lower fitness is assigned to it. Our method will select the individual which doesn't satisfy the outer condition but satisfies more inner conditions to generate the optimization solution. So the average number of generations of our method are higher than Kalaji's and the success ratio of our method is also higher than Kalaji's. Similarly, we can generate 12 paths for Transport Protocol EFSM in terms of the transition



(a) Simple in-flight safety system EFSM



(b) Class II transport protocol EFSM

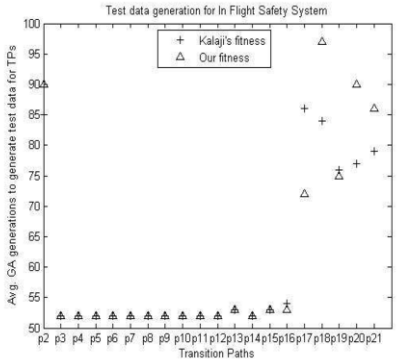


(c) Lift system EFSM

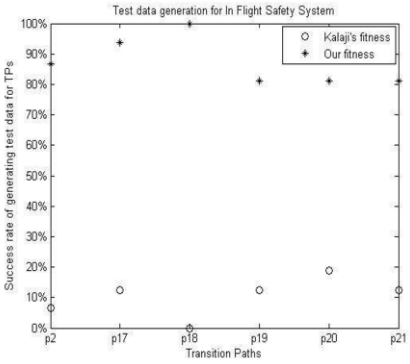
Fig. 1. EFSM examples

criterion. The average number of generations to generate test data for these paths are shown in Figure 2(c). Both methods can generate test data satisfying all the conditions for these paths except $p8$ and the average number of generations are almost the same. The reason why the two methods can only generate test data that do not satisfy all the conditions in the ten runs for $p8$ is the conditions in $p8$ include equation $Send\_sq == TRsq$. The range of equation just contains one value, the searching process of genetic algorithm can not converge, so genetic algorithm is hard to generate test data satisfying the equation.

There are 24 paths in Lift System EFSM. The genetic algorithm only can generate test data for $p1$ and $p2$. For other paths, the two methods both can only generate test data which do not satisfy all the conditions for these paths in each run, but the average number of generations of our method are higher than Kalaji's.



(a) Avg. generations to generate test data for in-flight safety system EFSM



(b) Success ratio of generating test data for in-flight safety system EFSM

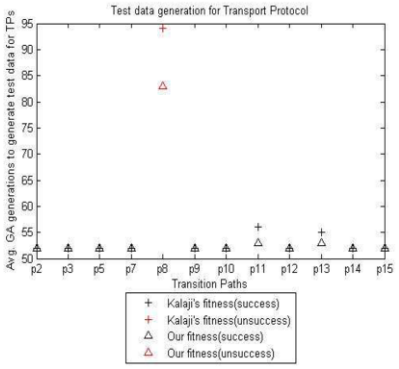
The reason is similar to the explanation in In Flight Safety System EFSM. In order to analyze the pros and cons of the two methods for these paths which can only get test data which do not satisfy all the conditions in each run, we compare Kalaji's method and ours by using the average ratio of conditions violated by the test data generated by the two methods.

The average violation ratio of conditions of Kalaji's method is more than 80%, and ours is approximately 10%, that is the test data generated by our method only violates the equation and the other conditions are all satisfied. We measure the quality of the test data generated by the two methods using average violation ratio of the conditions in the path, the lower the average violation ratio of the test data is, the higher the quality of it has. Seeing from Figure 2(e), the quality of the test data generated by our method is much better than Kalaji's.

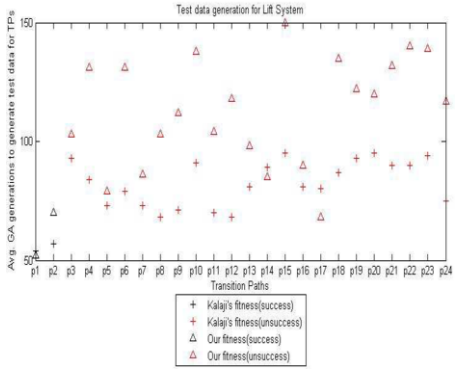
We take the test data generated for path $p3$ as example. The test data generated by Kalaji's and our method in one run are showed in table 2. The path constraint is $Pos \geq 0\ AND\ Pos \leq 15\ AND\ Pos1 \geq 0\ AND\ Pos1 \leq 15\ AND\ w \geq 15\ AND\ w \leq 250\ AND\ Pf == 1\ AND\ Ph \geq 10\ AND\ Ph \leq 35\ AND\ Ps \geq 0\ AND\ Ps \leq 2$. There are 11 conditions in the path constraint, and the average number of the conditions violated by the ten groups of test data generated by Kalaji's method is 10, so the average violation ratio is about 91%. Only the condition involving variable w is satisfied. Because the range of variable w is [15,250], the search scope is larger, and the condition can be satisfied easier. The average violation ratio of ten groups of the test data generated by our method is 1.2, that is the average violation ratio is 11%. All the value of the variables except the variable including in the equation meet the conditions.

Of course, Kalaji's method expresses the constraint as nested form, and our method represents the constraint as parallelism form, when the outer condition is not satisfied, Kalaji's method computes the fitness ignoring the inner conditions, and our method computes the fitness until all the conditions are analyzed. So our method is slower than Kalaji's.

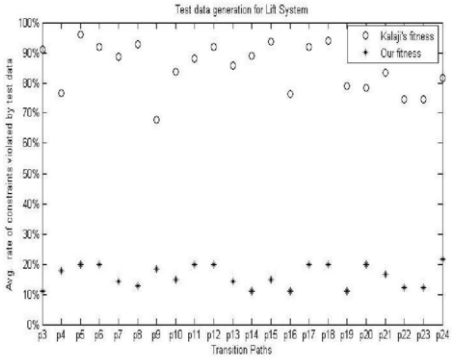
In Kalaji's paper, the search-based method has been compared with constraint-based method, and the constraint method is faster than search-based method, so



(c) Avg. generations to generate test data for Class II transport protocol EFSM



(d) Avg. generations to generate test data for Lift system EFSM



(e) Avg. violation ratio of conditions of Lift system EFSM

Fig. 2. The experimental results of the EFSM examples

we don't compare our method with constraint-based method.

6 Conclusions

This paper proposes an approach to generate test data for EFSM paths considering condition coverage. We analyze the Kalaji's fitness function, find out some problems of it, then give a new fitness function combining the branch distance with uncovered condition ratio, and compare our proposed method with Kalaji's from three aspects using three EFSM models:

(i) The average number of generations of our method to generate test data for the paths only involving simple constraint or single variable is the same as Kalaji's.

(ii) For the paths having more complex constraints or involving more variables, the success rate of our method to generate test data is higher than Kalaji's.

(iii) For the paths which include equation, both methods can only generate test data which do not satisfy all the conditions for them in each run. We use the average violation ratio of the conditions of the test data to measure the quality

Table 2
The test data generated for path p3

Methods	Test inputs(variables $pos, pos1, w, pf, ph, ps$)
Kalaji's method	15.39133,62.62504,-15.91648,38.15217,-27.55513,218.85892
	15.03939,193.80234,-194.5176,60.37151,-38.02112,-193.37181
	15.11819,-490.47645,-2.95238,230.43769,182.22669,165.32941
	15.92332,138.21063,228.69416,240.67647,-28.72298,-126.4982
	15.22626,468.96021,18.46922,272.68582,82.63248,315.80709
	12.92843,-0.7445,176.06993,50.38237,65.32962,-38.28198
	17.2368,22.55876,25.68157,67.54142,99.38773,65.66345
	15.31331,106.59772,-49.16128,135.67515,45.85847,210.19082
	15.14702,-218.11485,-31.06423,44.1253,355.31375,7.57338
	15.75495,56.36216,0.49388,17.74789,94.31005,30.0455
Our method	13.0755,11.14832,98.09096,0.22852,16.54342,14.69253
	11.42778,12.57557,46.22755,6.93122,33.77576,22.61043
	6.37736,12.92,28.05086,-1.02062,24.39492,24.68039
	4.01898,9.33363,121.56839,1.95281,32.42047,3.63125
	2.83836,2.15866,158.08038,7.75823,18.10003,8.24453
	8.75266,11.77969,99.40202,4.88325,34.42408,5.56386
	0.51374,5.91561,38.51984,0.51857,11.3195,10.72043
	4.95459,13.66483,64.21061,1.68918,8.12128,3.31641
	9.10128,19.74321,15.68837,1.81516,18.73419,8.24555
	14.48655,4.13108,47.01741,6.09891,27.96837,24.43352

of the test data generated by the two methods, the quality of the test data generated by our method is much higher than the Kalaji's.

And finally, Kalaji's method expresses the constraint as nested forms, and our method represents the constraint as parallelism forms, so our method is slower than Kalaji's.

References

[1] Andrews, A. A., J. Offutt and R. T. Alexander, *Testing Web applications by modeling with FSMs*, Software and System Modeling. **4** (2005), 326–345.

[2] Yang, R., Z. Y. Chen and B. W. Xu, et al, "Improve the Effectiveness of Test Case Generation on EFSM

via Automatic Path Feasibility Analysis," In Proc. of the 2011 IEEE 13th International Symposium on High-Assurance Systems Engineering (HASE 2011). IEEE Computer Society, (2011), 17–24.

[3] Gotlieb, A., B. Botella and M. Rueher, *Automatic Test Data Generation using Constraint Solving Techniques*, ACM SIGSOFT Software Engineering Notes. **23** (1998), 53–62.

[4] McMinn, P., *Search-based Software Test Data Generation:A Survey*, Journal of Software Testing,Verification and Reliability. **14** (2004), 105–156.

[5] Kalaji, A. S., R. M. Hierons and S. Swift, *An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models*, Information and Software Technology. **53** (2011), 1297–1318.

[6] Gupta, N., A. P. Mathur and M. L. Soffa, *Automated Test Data Generation Using An Iterative Relaxation Method*, ACM SIGSOFT Software Engineering Notes. **23** (1998), 231–244.

[7] Korel, B., *Automated Software Test Data Generation*, IEEE Transaction on Software Engineering. **16** (1990), 870–879.

[8] Ferguson, R., and B. Korel, *The Chaining Approach for Software Test Data Generation*, ACM Transactions on Software Engineering and Methodology. **5** (1996), 63–86.

[9] Miller, J., M. Reformat and H. Zhang, *Automatic test data generation using genetic algorithm and program dependence graphs*, Information and Software Technology. **48** (2006), 586–605.

[10] Ayari, K., S. Bouktif and G. Antoniol, " Automatic Mutation Test Input Data Generation via Ant Colony," In Proc.of the 9th Annual conference on Genetic and evolutionary computation. (2007), 1074–1081.

[11] Blanco, R., J. Tuya and B. Adenso-Daz, *Automated test data generation using a scatter search approach*, Information and Software Technology. **51** (2009), 708–720.

[12] Pargas, R. P., M. J. Harrold and R. R. Peck, *Test-Data Genenration Using Genetic Algorithms*, Journal of Software Testing, Verification and Reliability. **9** (1999), 263–282.

[13] Zhang, J., C. Xu and X. L. Wang, " Path-Oriented Test Data Generation Using Symbolic Execution and Constraint Solving Techniques," In Proc. of the Second International Conference on Software Engineering and Formal Methods (SEFM 2004). IEEE Computer Society, (2004), 242–250.

[14] Lefticaru, R., "Automatic State-Based Test Generation Using Genetic Algorithms," In Proc. of 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. (2007), 188–195.

[15] Lefticaru, R., F. Ipate, "An Improved Test Generation Approach from Extended Finite State Machines Using Genetic Algorithms," In Proc. of 10th International conference on Software Engineering and Formal Methods (SEFM 2012). (2012), 293–307.

[16] 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 Proc. of 10th Annual IEEE International High-Level Design Validation and Test Workshop. (2005), 87–93.

[17] King, J. C., *Symbolic Execution and Program Testing*, Communications of the ACM. **19** (1976), 385–394.

[18] Wiser, M., *Program Slicing*, IEEE Transaction on Software Engineering. **10** (1984), 352–357.

[19] Liu, S. M., Z. G. Wang, *Genetic Algorithms and its Application in path-oriented test data automatic generation*, Procedia Engineering. **15** (2011), 1186–1190.