

Automatic State-Based Test Generation Using Genetic Algorithms

Raluca Lefticaru, Florentin Ipaté
Department of Computer Science and Mathematics
University of Pitesti
Str. Targu din Vale 1, 110040 Pitesti, Romania
Email: raluca.lefticaru@gmail.com, florentin.ipate@ifsoft.ro

Abstract

Although a lot of research has been done in the field of state-based testing, the automatic generation of test cases from a functional specification in the form of a state machine is not straightforward. This paper investigates the use of genetic algorithms in test data generation for the chosen paths in the state machine, so that the input parameters provided to the methods trigger the specified transitions.

Keywords: Automated test data generation, finite state machines, search-based software engineering, evolutionary testing, genetic algorithms, mutation testing

1 Introduction

State machine diagrams (or statecharts before UML version 2.0) have been widely employed by many modelling languages and development methodologies, most importantly UML [4], for modelling the state behaviour of objects. State diagrams are formally described by finite state machines (FSMs), for which powerful test selection methods exist [7, 12]. These methods have also been used for testing state-based programs, but the transformation of the function sequences (from the state diagram) into methods call sequences is hindered by the difficulty of choosing the input values to be provided to the methods.

Many papers in the field of software testing define coverage criteria for state machines [7, 20, 6, 30], present the generation of test cases from UML state diagrams and evaluate their fault detection capability [20, 1, 21, 5]. Given a state machine diagram, it is possible to automatically derive a transition tree and obtain test sequences that accomplish certain criteria [21]. Briand et al. used test cases manually derived from UML state diagrams to evaluate their fault detection capability, identified their weaknesses and proposed ways to address them by combining other test strategies with state-based testing [1, 5]. Other authors have obtained test cases using statistical functional testing [6] or with the

aid of artificial intelligence methods, by mapping state diagrams to a planning language and then a planning tool (*graphplan*) [8]. Some tools, capable of generating actual test values automatically, also exist [20, 6, 21]. However, the process of developing such kind of tools is very complex and some limitations can appear.

The approach we propose is to employ genetic algorithms to produce the actual test values for feasible sequences of methods derived from the state diagram of the object, having the transition guards described in terms of algebraic predicates.

At same time, there has been much research in the area of search-based testing (especially for structural testing) in order to automatically generate input data for programs written mainly in a procedural paradigm [25, 22, 23]. Evolutionary methods for testing state-based programs have also been studied, but only from a structural testing point of view [15, 16]. Functional search-based testing has been less investigated and the studies have concentrated on Z specifications [11, 24, 26]. The test generation strategy we present in this paper, of using genetic algorithms on a state machine model of the system, is an evolutionary approach that has not been proposed before, to the best of our knowledge.

2 State-based testing

*Finite state machines (FSMs), or automata, are simple mathematical means of describing state diagrams. A FSM A consists of a finite set of states Q , of which one is the designated initial state q_1 , and transitions between states labelled by symbols from a finite alphabet X [7]. The FSM model of a simplified *Book* class from a library system is given in Fig. 1(a) and will be used for illustrations. The model is sufficiently self explanatory and so, due to space constraints, we will not provide further details.*

Once we have a FSM representation of a system, we can use appropriate techniques to uncover possible errors in the implementation, such as erroneous transition labels, erroneous next-states, missing states, extra states, etc. One of

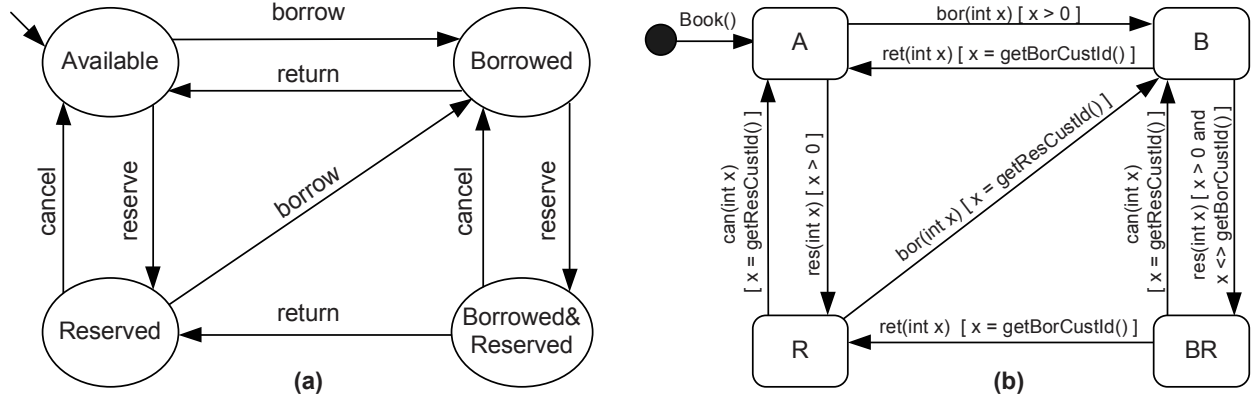


Figure 1. State model of a Book class: (a) FSM (b) State machine diagram

the most general approaches is the W -method [7], that generates sequences of symbols to reach every state in the diagram, check all the transitions from that state and identify all destination states to ensure that their counterparts in the implementation are correct. Thus the W -method involves the construction of two sets of sequences:

- A *transition cover* $T \subseteq X^*$ that reaches every state of the FSM and exercises every transition from that state. T can be written as $T = S \cup SX$ for some set of sequences $S \subseteq X^*$, called a *state cover*, that reaches every state of the FSM.
- A *characterization set* $W \subseteq X^*$ that distinguishes between every pair of states in the FSM. In other words, for every pair of distinct states q_1 and q_2 , W contains at least one sequence of labels that can be applied from q_1 but not from q_2 or vice-versa.¹

In Fig.1(a), the empty sequence ϵ reaches the initial state *Available*, *borrow* reaches *Borrowed*, *reserve* reaches *Reserved*, while *borrow reserve* reaches *Borrowed&Reserved*. Thus $S = \{\epsilon, borrow, reserve, borrow reserve\}$ is a state cover and $T = S \cup SX$ is a transition cover. On the other hand, it can be observed that *return* partitions the state space into $\{Available, Reserved\}$ and $\{Borrowed, Borrowed\&Reserved\}$, while *cancel* distinguishes between the elements of each of the two classes of the partition. Thus, $W = \{return, cancel\}$ is a characterization set. In general, a transition cover is derived from a transition tree constructed in a breadth-first fashion, while a characterization set is constructed by gradually partitioning the state set based on the responses produced by sequences of length $k \geq 1$, using the so-called P_k tables [7].

¹A FSM in which all states are reachable and pairwise distinguishable is called *minimal*. This is not a restriction, however, since, for every FSM, a minimal, functionally equivalent, FSM can be constructed [12].

Suppose we have a FSM specification A and we have constructed a transition cover T and a characterization set W of A . Naturally, we assume that the implementation of A can be modelled by an unknown FSM A' . The only information we need about A' is an estimation of the maximum possible number of states n' that it may have. Suppose we denote by k , $k \geq 0$, the difference between this estimated upper bound and the number of states n of the specification A . Then the test suite produced is

$$Y = TX[k]W.$$

This formula is given in [7] for the case in which the FSM specification is completely-specified and transitions are labelled by outputs as well as inputs. As explained in [10], when the specification is partially-specified (the case considered in this paper), the test suite will also include some of the prefixes of the sequences in Y . However, in order to check the result produced by an input sequence, one would normally check the results produced by all its prefixes; consequently, this extra sub-set of prefixes does not explicitly appear in the formula.

The idea is that T ensures that all the states and all the transitions in A are also present in A' and $X[k]W$ ensures that A' is in the same state as A after performing each transition. Note that the latter set contains W and also all sets X^iW , $1 \leq i \leq k$. This ensures that A' does not contain extra states or, if there are some extra states, they behave the same way as the corresponding specification states. If there were up to k extra states, then each of them would be reached by some input sequence of length up to k from the existing states.

Variants of the W -method also exist in the literature. The round trip [3] approach is based on a transition tree constructed in a depth-first fashion. Furthermore, an abstract oracle [17] can be used instead of W as in [5] to check the state of the implementation under test.

3 Genetic algorithms

Genetic algorithms (GAs) [29, 19] are a particular class of *evolutionary algorithms*, that use techniques inspired from biology, such as selection, recombination (crossover) and mutation. They were conceived by John Holland in the United States in the late sixties and are closely related to evolution strategies, developed independently at about the same time in Germany by Ingo Rechenberg and Hans-Paul Schwefel.

GAs are used for problems which cannot be solved using traditional techniques and for which an exhaustive search of the solution space is impractical, by encoding a population of potential solutions on some data structures, called *chromosomes* (or *individuals*) and applying recombination and mutation operators to these structures. A high level description of a genetic algorithm [14] is given in Fig. 2. The *fitness (objective) function* assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how close that chromosome is to the solution of the problem. Throughout this paper, the fitness is considered to be positive and so finding a solution corresponds to minimizing the fitness function, i.e. a solution will be a chromosome with fitness 0. The algorithm terminates when some stopping criterion has been met, for example when a solution is found, or when the number of generations has reached the maximum allowed limit.

Various mechanisms for selecting the individuals to be used to create offspring, based on their fitness, have been devised [9]. Holland's original GA used fitness-proportionate selection, in which the "expected value" of an individual (i.e. the expected number of times an individual will be selected to reproduce) is that individual's fitness divided by the average fitness of the population. This kind of selection leads to "premature convergence". To address such problems, GA researchers have experimented other mechanisms such as sigma scaling, elitism, Boltzmann selection, tournament, rank and steady-state selection [19].

After the selection step, recombination takes place to form the next generation from parents and offspring. Single-point crossover, probably the best known form of recombination, randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two new offspring. For example, the strings 00000000 and 11111111 could be crossed over at the third locus to produce the two offspring 00011111 and 11100000. Crossover is applied to individuals selected at random, with a probability (rate) p_c . Depending on this rate, the next generation will contain the parents or the offspring.

The mutation operator randomly flips some bits in a chromosome. For example, the string 00000100 could be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probabil-

Randomly generate or seed initial population P

Repeat

Evaluate fitness of each individual in P

Select P' from P according to selection mechanism

Recombine parents from P' to form new offspring

Mutate P'

$P \leftarrow P'$

Until Stopping Condition Reached

Figure 2. Genetic Algorithm

ity p_m , usually very small [19]. This operator is responsible for introducing variation in the population.

4 Test data generation strategy

The generation technique we propose is based on the state diagram of the object under test and uses a genetic algorithm to search for the test data (input parameters) which satisfy some specified requirements.

As powerful test selections methods for FSMs exist (and also a number of coverage criteria), it is normal to use these state-based models in class testing. The test set will be built from sequences of transitions obtained from the state machine, providing them with appropriate input parameters. A genetic algorithm will be used to search for test data (input values) which satisfy the requirements of each transition.

The first step of the test generation is to obtain some paths in the state machine, according to some coverage criteria (such as all transitions, full predicate, transition pair, complete sequence or disjunct coverage [7, 3, 20, 21, 30]). Once the state machine of the class is available (with guard conditions included in the transition labels), it is possible to automatically derive a transition tree and obtain test sequences that accomplish these criteria [21].

The second step is finding, for each sequence, the input parameter values which trigger the methods in the sequence.

In order to illustrate our approach we will consider a state machine representation of a *Book* class, having the transitions labelled with the method names, including their parameter list and guard conditions, like in Fig.1(b). The states and the method names from Fig. 1(b) are acronyms for their correspondents in the FSM representation of the class from Fig. 1(a). Each method may appear more than once per state if the guard conditions are different ². The *Book* class has public getters for the customer which has reserved the book and for the customer which has borrowed the book: *getResCustId()* and *getBorCustId()*, respectively.

²Note that in this state diagram, the erroneous transitions are missing. They are in fact self-loops which do not change the state of the object and for which the guard is the negation of the correct usage of the method

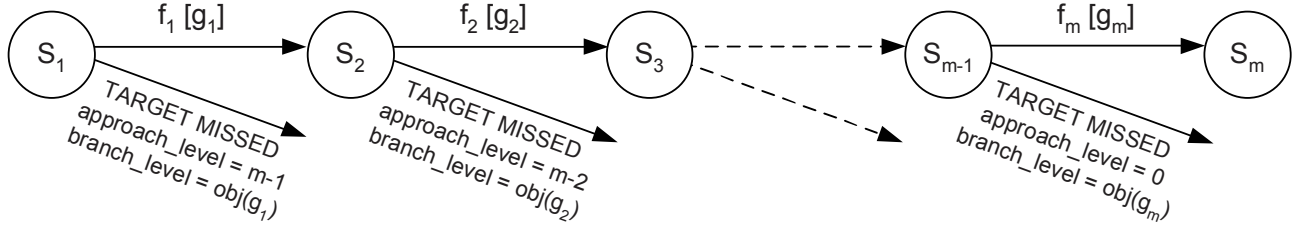


Figure 3. Calculating the fitness function

The guards express conditions related to the the input parameters and/or the current "memory" of the machine, like: $bor(x)[x = getResCustId()]$ transition from state R to state B or $bor(x)[x <> getResCustId()]$ transition from R to R (corresponding to a forbidden usage of the method).

4.1 Chromosomes

Given a particular path in the state machine, like $A \xrightarrow{bor(x_1)} B \xrightarrow{res(x_2)} BR \xrightarrow{ret(x_3)} R \xrightarrow{bor(x_4)} B$ and the corresponding constraints for each transition in the path, a *chromosome* (possible solution) is a list of input values, for example $x = (x_1, x_2, x_3, x_4)$, corresponding to all parameters of the path methods (in the order they appear). If the sequence of method calls with parameter values from a chromosome x determines the transitions between the states specified by the path and validate the predicate of each transition, then x is a solution for the given path.

The transitions constraints are not necessary the guards from the state machine diagram; they can be predicates obtained from stronger variants of state-based criteria, such as full predicate [21], in which all individual clauses in a decision are exercised, or disjunct coverage [5], in which guard conditions are transformed into disjunctive normal form and a separate transition is defined for each disjunct. For example, in order to achieve full predicate coverage, for each predicate P on each transition and each test clause c in P , the test set must include tests that cause each clause c in P to *determine* the value of P , where c has both true and false values [21]. If the predicate P is $X \vee Y$ and the test clause is X , then Y must be false; similarly, if the test clause is Y then X must be false. Then, for the predicate $X \vee Y$, the test set must include test cases satisfying: $X \wedge \bar{Y}$, $\bar{X} \wedge Y$ and $\bar{X} \wedge \bar{Y}$. More details are given in [20, 21].

Every individual (chromosome) is represented as $x = (x_1, x_2, \dots, x_n) \in D_1 \times D_2 \times \dots \times D_n$, where D_1, D_2, \dots, D_n are the domains of the input variables from the method sequence considered. It is obvious that n can differ from the number of functions in the sequence and the domains D_1, D_2, \dots, D_n might be distinct.³

³The class constructors can appear like any other method in the se-

Relational predicate	Objective function obj
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
Connective	Objective function obj
Boolean	if $TRUE$ then 0 else K
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
$a \text{ xor } b$	$obj((a \wedge \bar{b}) \vee (\bar{a} \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

Table 1. Tracey's objective functions for relational predicates and logical connectives. The value K , $K > 0$, refers to a constant which is always added if the term is not true

The algorithm evaluates each individual by creating an instance of the class under test and executing each method call with the values encoded in the chromosome's genes x_i , until a method's constraint is not satisfied. The fitter individuals are the ones which follow more transitions from the given path (i.e. the ones which accomplish more transitions guards) and so they are rewarded with lower fitness values.

4.2 Fitness function

The fitness value of one individual will be made up of two components. The first will evaluate how close a chromosome is to the given path, by counting the transitions executed. The second component will measure how close was the first unsatisfied pre-condition predicate to being true.

quence, but for simplicity we considered in this example a default constructor which brings the object in the initial state A .

Function Fitness

Input: Target path s , containing the methods $f_1, f_2 \dots f_m$
Corresponding guards g_1, g_2, \dots, g_m
Chromosome $x = (x_1, \dots, x_n)$

Begin

Create an object o in the initial state

$approach_level \leftarrow m - 1$

For every method f_i in the sequence s ($i = 1..m$) do

If *not* g_i

Calculate $obj(g_i)$

Return $approach_level + norm(obj(g_i))$

Else

$approach_level \leftarrow approach_level - 1$

Call the method f_i of object o with the corresponding values from (x_1, \dots, x_n)

End If

End For

Return 0

End

Figure 4. Fitness Evaluation

The first component of our fitness function has a similar metric in evolutionary structural test data generation, the *approach (approximation) level*. This is calculated by subtracting one from the number of critical branches lying between the node from which the individual diverged away from the target node, and the target itself. In this approach, a critical branch is a program branch which leads to a miss of the current structural target for which test data is sought [16]. In our case we consider all the transitions to be critical (for the transitions without a guard, we consider the *TRUE* predicate as pre-condition).

An individual has missed the target path if at some point a pre-condition is not satisfied (this could cause a transition to other state). The *approach_level* will be 0 for the individuals which follow the target path; for those which diverge from the path, it will be calculated as described in Fig. 3.

Because a fitness function employing just the approach level has many plateaux (for each value $0, 1, \dots, m - 1$), it does not offer enough guidance to the search. Consequently, we will also need a second component of the fitness function. This will compute, for the place where the actual path diverges from the required one, how close was the pre-condition predicate to being true. This second metric is called *branch level* and can be derived from the guard predicates using the transformations from Table 1 [24, 26, 14]. The branch level is then mapped then onto $[0, 1)$ - is said to be *normalized*. Thus the fitness value is:

$$fitness = approach_level + normalized_branch_level$$

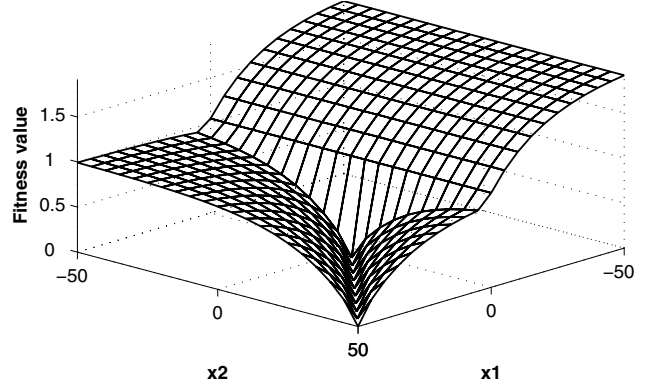


Figure 5. Fitness function landscape

For every chromosome (x_1, x_2, \dots, x_n) , the fitness value will be computed dynamically, as described in Fig. 4.

In order to illustrate the fitness landscape, consider as target path the sequence $s = A \xrightarrow{res(x_1)} R \xrightarrow{bor(x_2)} B$ which has the derived pre-conditions: $x_1 > 0$ and $x_2 = x_1$ ⁴. A fitness function for this sequence is represented in Fig. 5, where the constant from Tracey's objective function considered was $K = 1$ and the normalization function was $norm : [0, 101] \rightarrow [0, 1), norm(d) = 1 - 1.05^{-d}$.

5 Related work

Metaheuristic search techniques have been applied for automatic generation of test data for structural and functional testing, grey-box and non-functional properties testing. A comprehensive survey on the search-based data generation is [14].

Evolutionary testing was assessed in the context of structural testing in many papers like [22, 26, 23]. The state problem was analyzed from the structural point of view, regarding the flags (loop-assigned flags), enumerations and counters used to manage the internal state of the objects in [2, 15, 16]. Generation of test data from a Z specification was presented in [11] and extended in [24, 26].

In [23] Tonella presents a genetic algorithm employed to produce test cases with the aim of maximizing a given coverage measure. The test cases are described by chromosomes, which include information on which objects to create, which methods to invoke and which values to use as inputs. A set of appropriate mutation operators are also defined.

The specification conformance issue is addressed in [24, 26]. For this, a constraint system was derived by tak-

⁴In order to obtain $x_2 = x_1$, we used the post-condition $getResCustId() = x_1$, representing a correct usage of $res(x_1)$ method.

ing the pre-condition in conjunction with the negated post-condition. The landscape of the fitness function obtained, $obj(\text{pre-condition}) \wedge obj(\neg \text{post-condition})$, contained areas of plateaux. The solution found to offer more guidance to the search was to convert the constraint system to the disjunctive normal form and consider each disjunctive in a separate search.

6 Empirical verification

Experiments were performed for two Java classes, using the test data generation approach presented in section 4. For implementing the basic genetic mechanisms we used JGAP (Java Genetics Algorithm Package). More details about this genetic algorithms and genetic programming framework for Java are available on the JGAP web site [27].

6.1 Experiment 1

The first experiment consisted in generating test data for a *Book* class, having the state-machine diagram from Fig. 1(b). Several method sequences, with different lengths and constraints, were used. The fitness function was calculated using the pre-conditions, as described in section 4.2. Although the example is small (121 lines of code), the complexity of the search-based testing depends on the number of solutions from the search space, not on the program size.

An elitist genetic algorithm was used in the experiments, with the following default configuration values: a population size of 20 individuals, maximum allowed evolutions = 200; the integer values were restricted to $[-10^p, 10^p]$, $p \in \{1, 2, 3, 4\}$. Thus, the search space of a sequence involving n method parameters, $p = 4$, was approximately $2^n \times 10^{4n}$.

The chromosomes were real-encoded, each gene representing one input parameter. The operators used from JGAP framework were the *BestChromosomesSelector*, with a 0.8 rate (this parameter controls how many chromosomes of the original population will be considered for selection to the next population) and the *MutationOperator*, with a default 1/15 mutation rate. Unlike the binary mutation operator, which flips some bits (genes), the one used for real encoding replaces a gene value with another real value from the same interval, randomly obtained.

For recombination, we tested several operators and the most efficient for our problem was the heuristic real value crossover inspired from [18], which uses values of the objective function in determining the direction of the search. For the parents $x = (x_1, \dots, x_n)$, $y = (y_1, \dots, y_n)$, x fitter than y , one offspring $z = (z_1, \dots, z_n)$ was generated, with $z_i = \alpha \cdot (x_i - y_i) + x_i$, $\alpha \in (0, 1)$. Instead of $\alpha \in (0, 1)$ randomly chosen, better scores were obtained for a fixed $\alpha = 0.2$. It is worth noting that the usage of

other recombination operators (like single-point crossover) when the input values $x = (x_1, \dots, x_n)$ were highly correlated, caused many failures of the algorithm, because the offspring obtained were losing the relations between the values x_1, \dots, x_n and so they were less fit than the parents.

There are some interesting things to note about the experiment. In order to compare the difficulty of test data generation for the given paths, the length of a path (number of transitions) is not an appropriate measure. For example, for the path $A \xrightarrow{bor(x_1)} A \xrightarrow{res(x_2)} A \xrightarrow{ret(x_3)} A \xrightarrow{can(x_4)} A \xrightarrow{ret(x_5)} A$ it is very easy to find test data even at a random search, since the transitions constraints are: $x_1 \leq 0$, $x_2 \leq 0$, x_3, x_4, x_5 can have arbitrary values; the parameters x_1, x_2, x_3, x_4 and x_5 are not correlated and, consequently, the solution space represents around 1/4 of the entire search space. On the other hand, for the

path $A \xrightarrow{bor(x_1)} B \xrightarrow{res(x_2)} BR \xrightarrow{ret(x_3)} R \xrightarrow{can(x_4)} A$, the derived constraints are $x_1, x_2, x_3, x_4 > 0$, $x_2 \neq x_1$, $x_3 = x_1$, $x_4 = x_2$. Thus, the probability to randomly find a solution for this path is approximately $1/(2^4 \cdot 10^8)$, when $x_i \in [-10^4, 10^4]$.

In order to evaluate the complexity of data generation for different paths, we chose 35 paths with lengths between 2 and 7, calculated the probability of randomly finding a solution for every path (according to its pre-conditions and parameter search space - the *Ox* axis in Fig. 6) and recorded the experimental results obtained in generating data for every path. We varied the ranges of the parameters (resulting a different dimension of search space and path solution probability) and represented in Fig. 6 the average number of generations needed to obtain a solution for a path (calculated after 1000 executions of the program) and the percentage of times the program failed to find a solution for the path.⁵ Similar results were obtained for paths corresponding to a Java *VendingMachine* implementation.

6.2 Experiment 2

The second experiment focused on testing the conformance of the implementation to its specification. While in the first experiment we only generated data from the state-machine specification, in the second we also validated the implementation against the specification.

In order to verify the correctness of the implementation, the fitness function calculation can be modified, to check after every method call (when the guard is satisfied) if the post-condition holds. The post-condition can be represented by the expected state, the value returned by

⁵For the Fig. 6 all the paths having the solution probability in the interval $[10^{-p-1}, 10^{-p}]$, $p \geq 0$ were grouped and only the average probability of the paths, with the average number of generations and the average percentage of failures, respectively, were plotted.

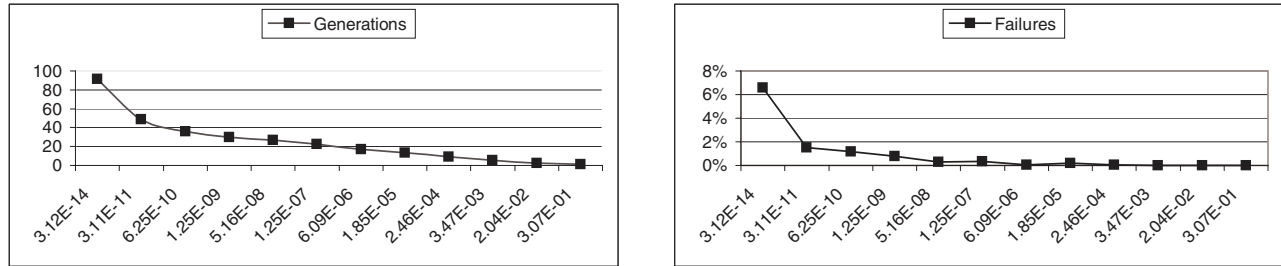


Figure 6. Variation of the average number of generations and percentage of failures for some Book sequences, related to: $solutions_number / possible_solutions_number$ (Ox axis)

the method called and other expected changes. In order to check the current state, an abstract oracle [17] can be used like in [5]. The state invariants can be expressed using the class methods (for example, $getResCustId() > 0$ and $getBorCustId() = 0$ corresponds to the *Reserved* state) or the post-condition can be more explicit (for example, $getResCustId() = x_1$ after a successful method call $res(x_1)$).

In our experiment, a constraint system was build for the *Book* class under test, having the pre-conditions from the state machine representation and the post-conditions expressed in terms of: expected state after transition, true/false results of the method call for successful/unsuccessful realization, modifications of the current borrow/reserve customer id. The fitness function was derived to measure how "close" a failure is, by taking the pre-condition in conjunction with the negated post-condition of the last transition.

In order to evaluate this approach, the genetic algorithm implementation employing the redesigned fitness function was run against mutated versions of the *Book* class, using 24 transition sequences. These sequences were obtained from a transition tree of the state machine, build in a breadth-first fashion [7] and had a maximum length of 3. For every mutant, they were fed to the program in order of their lengths.

The mutated class versions were generated automatically by MuJava [13, 28], a mutation system for Java programs. In the experiment we considered all the traditional mutants of the methods *bor*, *res*, *ret* and *can* of the *Book* class.

For every mutant version of the *Book* class, the genetic algorithm procedure took each sequence and searched for data capable of minimizing the fitness function. Once a post-condition was violated with the pre-condition satisfied, or the invoked method produced an exception, the procedure recognized the mutant and recorded the sequence with the input values which uncovered the fault. If no solution to minimize the fitness function was found for any of the sequences, the best value obtained thus far was also recorded

and input data was searched for the next mutant.

The experiment was realized having the maximum allowed evolutions set to 100 and a population size of 20 individuals. Under these conditions, 103 mutants from a total of 125 were discovered. The maximum number of allowed evolutions was increased ⁶ and run again only against the undetected mutants. 7 additional mutants were killed, while the remaining 15 were checked and found to be equivalent to the original code. The big number of evolutions needed to discover the 7 remaining non-equivalent mutants was due to the fact that the fitness function did not offer enough guidance to the search. This is because a negated post-condition, like $x \langle \rangle getBorCustId()$ is transformed to $norm(K)$ for the false branch, and consequently the fitness landscape will contain plateaux.

To compare the power of generating significant data, a set of test cases were manually written for the same 24 transition sequences and run against the mutants.

The results were: 103+7 mutants discovered using the genetic algorithm and 100 by the manually derived test cases. One difference was due to the relational operator mutants⁷. Consider, for example, a simple guard on a transition (or clause in a guard) like $x \langle \rangle value$. Other transition from the same state will have the negated guard $x = value$. Then, one of the two mutants in which the guard $x \langle \rangle value$ has been replaced by $x < value$ and $x > value$ respectively, will not be killed using a single test value for the clause $x \langle \rangle value$. This was the case for manually derived test sequences, that covered each transition guard with only one value. On the other hand, the genetic algorithm obtained for each mutant the appropriate input values to discover it and of course these values were different from one mutant to another, even when the transition sequence was the same. Some mutants were related to guards like $x \leq 0$ which were tested only by $x < 0$ or

⁶Another way to achieve a better exploration of the search space is to decrease the range of the parameter values, which was $[-10^4, 10^4]$

⁷This mutation operator replaces a relational operator $==, <, \leq, >=, >, !=$ with another one

$x = 0$ and one mutant remained undetected for the prepared test suite, but not for the genetic algorithm implementation.

An additional case study examined the fault detection capacity of the test data generated using the state-based approach and genetic algorithms against the mutants of the *VendingMachine* class. Using the same configuration values (maximum allowed evolutions 100, population size 20, parameter range $[-10^4, 10^4]$) 72% of mutants were killed. After increasing the maximum evolutions number, 5% additional mutants were uncovered. The remaining 13% were found to be equivalent with the original code (and satisfying the specification requirements) while 10% were related to private attributes of the class, not present in the state machine diagram. These mutants could have been killed only with a precise oracle or using additional post-conditions of the methods.

7 Conclusions and future work

This paper presents an approach for automatic generation of test data, using state diagrams and genetic algorithms. The strategy is simple, the derivation of the fitness function is straightforward and so could be easily adopted in industrial software development.

Furthermore, experimental evidence show that the test data obtained can cover difficult paths in the machine and a slightly different design of the fitness function can be used for specification conformance testing.

Future work concerns the possibility of extending the strategy presented for multi-class testing and the derivation of the fitness function from hierarchical and concurrent state machine diagrams.

References

- [1] G. Antoniol, L. C. Briand, M. D. Penta, and Y. Labiche. A case study using the round-trip strategy for state-based class testing. In *ISSRE '02*. IEEE Computer Society, 2002.
- [2] A. Baresel, D. Binkley, M. Harman, and B. Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *ISSTA*, pages 108–118, 2004.
- [3] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology*. Addison-Wesley, 1999.
- [4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.
- [5] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Softw. Eng.*, 30(11):770–793, 2004.
- [6] P. Chevalley and P. Thévenod-Fosse. Automated generation of statistical test cases from UML state diagrams. In *COMP-SAC '01*, pages 205–214. IEEE Computer Society, 2001.
- [7] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.
- [8] P. Fröhlich and J. Link. Automated test case generation from dynamic models. In *ECOOP '00*, pages 472–492. Springer-Verlag, 2000.
- [9] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, pages 69–93, 1990.
- [10] F. Ipate. Testing against a non-controllable stream X-machine using state counting. *Theoretical Comput. Sci.*, 353(1-3):291–316, 2006.
- [11] B. Jones, H. Sthamer, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of 3rd International Conference on Software Quality Management*, pages 435–444, 1995.
- [12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
- [13] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.
- [14] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.
- [15] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *GECCO*, pages 2488–2498, 2003.
- [16] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *GECCO*, pages 1013–1020, 2005.
- [17] B. Meyer. Design by contract. *IEEE Computer*, 25(10):40–51, 1992.
- [18] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.
- [19] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998.
- [20] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML*, pages 416–429, 1999.
- [21] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.
- [22] R. P. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Softw. Test., Verif. Reliab.*, 9(4):263–282, 1999.
- [23] P. Tonella. Evolutionary testing of classes. In *ISSTA*, pages 119–128, 2004.
- [24] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98*, pages 73–81. ACM Press, 1998.
- [25] N. Tracey, J. A. Clark, K. Mander, and J. A. McDermid. An automated framework for structural test-data generation. In *ASE*, pages 285–288, 1998.
- [26] N. J. Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York, 2000.
- [27] <http://jgap.sourceforge.net/>. JGAP home page.
- [28] <http://www.ise.gmu.edu/~offutt/mujava/>. MuJava home page.
- [29] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.
- [30] Y. Wu, M.-H. Chen, and J. Offutt. UML-based integration testing for component-based software. In *ICCBSS*, pages 251–260, 2003.