Combining Genetic Algorithms and Mutation Testing to generate test sequences [*]

Carlos Molinero, Manuel Núñez and César Andrés

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
molinero@fdi.ucm.es, mn@sip.ucm.es, c.andres@fdi.ucm.es

Abstract. The goal of this paper is to provide a method to generate efficient and short test suites for Finite State Machines (FSMs) by means of combining Genetic Algorithms (GAs) techniques and mutation testing. In our framework, mutation testing is used in various ways. First, we use it to produce (faulty) systems for the GAs to learn. Second, it is used to sort the intermediate tests with respect to the number of mutants killed. Finally, it is used to measure the fitness of our tests, therefore allowing to reduce redundancy. We present an experiment to show how our approach outperforms other approaches.

1 Introduction

Software testing is an expensive and time consuming task. If a formal approach is used, tests are derived from a specification. For derived test sets to be complete, the tester needs to assume a given set of assumptions and hypotheses, allowing to reduce the space of the possible implementations. Our intention is to combine GAs and mutation testing to create a new approach capable of deriving a test suite that, with a short amount of execution time, finds the 95% of the faulty implementations. We call our methodology GAMuT (Genetic Algorithm and MUtation Testing) and it is composed of 3 main phases: Learning through evolution, learning through specialization, selection and reduction of the test cases.

GAs have shown to have a good performance in search and optimization problems. There exists a number of papers where GAs are used in testing (e.g.[4,5,2]). They usually represent the test data generation problem as an optimization problem and heuristics are used to generate test cases. Mutation Testing has been widely used for checking the performance in test suites, by measuring its capability to kill mutants, for details into applications developed through *mutation testing* see [3,8] for some formal approaches and [1] for a critical discussion on its use. We propose to use mutation as a way to provide the GA with enough learning examples in an automated way, by modifying the specification and subsequently creating simulations of faulty implementations. We compile several populations into a *community*. Therefore, each community has several *populations*, with several *inhabitants* each, and each individual has a DNA that directly

[*] Research supported by the Spanish projects WEST (TIN2006-15578-C02-01) and MATES (CCG08-UCM/TIC-4124).

represents a test sequence, that is, a chain of inputs that will be applied to the implementation under test (in short, IUT). In our methodology, mutation testing is used in three ways. In the first phase of GAMuT, called *learning through evolution*, the community is presented with a set of mutants. In our approach we use 100 IUTs and each of the populations is confronted with all the IUTs. The fitness function is a heuristic based on the percentage of mutants killed. The second phase of GAMuT, *learning through specialization*, also uses mutants but in a different way. In this case, each population inside the community (after having evolved to kill the biggest number of mutants) is given one IUT, mutated from the specification, and each population evolves to try and minimize the length of the sequence needed to find the error inside that mutant. Finally, we select the fittest individual from each population (from both phases) and create a set of tests, which we confront with the final and biggest set of mutants (500 in our methodology), to be able to find the number of mutants killed by each of the selected individuals. Then, as the last step, we make a subdivision with the smaller set of tests that kills the highest number of mutants.

The rest of the paper is organized as follows, in Section 2 we present the language used to define specifications, the mutation operators, and define the operations and objects of GA. In Section 3 we give a schema of the different phases of our methodology. In Section 4 we present experiments and the results of the comparisons of our technique with random testing. Finally, in section 5 we give our conclusions and some ideas for further development.

2 Preliminaries

We introduce hereby some general notions, about Finite State Machines, and about our GA. Specifications and its mutants are given by means of finite state machines.

Definition 1. A *Finite State Machine*, in short FSM, is a tuple $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_0)$ where S is a finite set of states, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, Tr is the set of transitions, and s_0 is the initial state. We say that M is *input-enabled* if for all state $s \in S$ and input $i \in \mathcal{I}$, there exist $s' \in S$ and $o \in \mathcal{O}$ such that $(s, s', i, o) \in Tr$. We say that M is *deterministic* if for all $s \in S$ and $i \in I$ there do not exist two different transitions $(s, s_1, i, o_1), (s, s_2, i, o_2) \in Tr$. $\square$

A transition belonging to Tr is a tuple (s, s', i, o) where $s, s' \in S$ are the initial and final states of the transition, and $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output actions, respectively. Intuitively, a transition (s, s', i, o) of a FSM indicates that if the machine is in state s and receives the input i, then the machine emits the output o and moves to s'. Along the rest of this paper we assume that all the machines are input-enabled and deterministic. The first restriction, as usual, is needed to ensure that implementations will react to any input provided to them. The second limitation is not vital to our approach, although it simplifies the implementation of the algorithm.

Next we define two mutation operators: One will modify the output of a certain transition and the other will modify the state at which the transition arrives.

Definition 2. Let $M = (S, I, O, Tr, s_0)$ be a FSM. The application of the operator $moper_1$ to M produces a mutant by choosing a transition $tr = (s, s', i, o) \in TR$ and an output $o' \neq o$, and by returning the FSM $M' = (S, I, O, Tr', s_0)$ such that $Tr' = Tr \backslash \{tr\} \cup \{tr'\}$, where $tr' = (s, s', i, o')$. The application of the operator $moper_2$ to M produces a mutant by choosing a transition $tr = (s, s', i, o) \in TR$ and a state $s'' \neq s$, and by returning the FSM $M' = (S, I, O, Tr', s_0)$ such that $Tr' = Tr \backslash \{tr\} \cup \{tr'\}$, where $tr' = (s, s'', i, o)$. $\square$

A usual problem in mutation testing is that the application of mutation operators may produce an *equivalent mutant*, that is, a mutant equivalent to the specification. Therefore, when applying mutation testing it is normal that tests do not kill all the mutants since some of them (the equivalent ones) are not even supposed to be killed.

GAs are an AI technique that uses metaphors of mechanisms present in nature for organism to develop, adapt and reproduce, to try to survive in the system in which they act and live. Its main operators are mutation, reproduction (genetic crossing) and selection of the fittest individuals. GAs are a good method to use with black-box testing since if we do not have any information regarding the internal structure of the IUT, then the testing problem can be expressed as a search problem, guided by a heuristic. In our case, the search space is the set of all possible implementations that may have mutated from the given specification. GAs can adapt themselves to find this optimum, as long as the fitness function is correctly defined. Let us note that if we were to leave a genetic algorithm running indefinitely, the whole population will converge to the same inhabitant, that would be the one that maximizes the fitness function.

3 Description of GAMuT

Next we describe our GA. Since we do not want to find a single solution and for the sake of genetic diversity, we have added another component to usual GAs: A *community* holds several *populations*, having in turn several inhabitants. We do not allow genetic crossing between populations. This can be seen as a parallelism to what is sometimes called in the literature *species*. An inhabitant of a specific population has a DNA sequence that is formed by genes, that codifies the test to be applied. The value that each gene can take is any input $i \in I$. The DNA sequence can be modified through mutation and recombination, that is, by mating of two inhabitants. In addition, it can also mutate the length of its sequence. The community holds the specification that we are trying to check as well as a set of examples of mutated specifications that we will call $\mathrm{ex}_{\mathrm{IUT}}$, and each population holds a mutated IUT specific for it, used in the specialization phase.

In order to initiate our algorithm, all genes are randomly initialized, so that a random number of gens is available for each possible DNA. The size of the population, the number of populations, and the fitness function are decisions to be made by the tester, that has to take into account the number of states of the machine. An example of fitness function used in GAMuT is shown in Section 4. The search finished by two means, one will be to reach the maximum number of generations specified for the GA, the other is that the test sequences are able to detect a specific percentage of the faulty IUTs. We count with two mutation operators for DNA: To change the value of one or more gens and to modify the length of the DNA sequence. DNAs can be modified also by combining them with the DNA of another inhabitant through mating. In our approach we have used a single point crossover, with random position for the division. As usual, we will use *elitism* as a way to control that only the best solutions actually get to mate and reproduce. We will use the *roulette wheel* selection technique that allows to choose the inhabitants in a proportional scale to its fitness. In our case, we also add after reproduction the original set of elected best inhabitants from the population, not to loose solutions. In order to decide who are the best, we find the top/lowest fitness scores, and we take a percentage of the top to be chosen as the best.

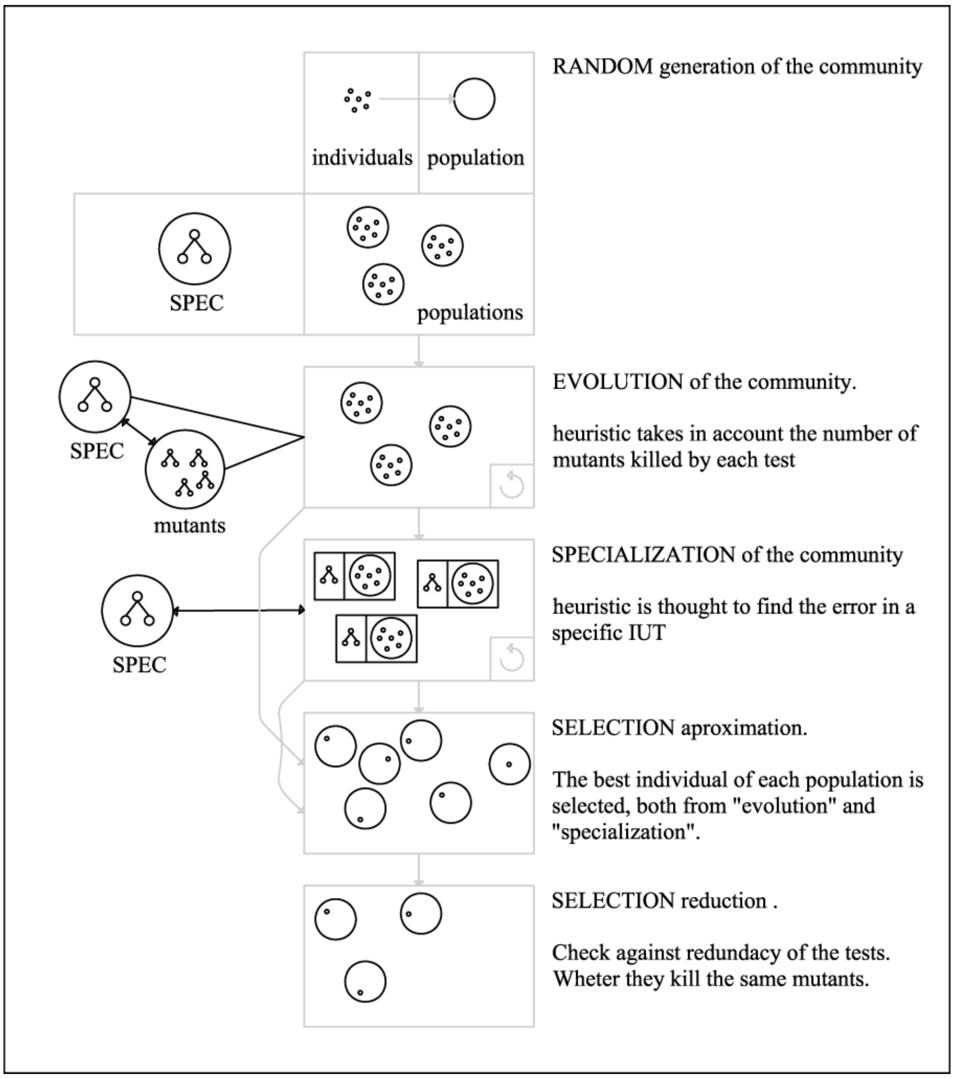
RANDOM generation of the community
individuals
population
SPEC
populations
EVOLUTION of the community.
heuristic takes in account the number of mutants killed by each test
SPEC
mutants
SPECIALIZATION of the community
heuristic is thought to find the error in a specific IUT
SPEC
SELECTION aproximation.
The best individual of each population is selected, both from "evolution" and "specialization".
SELECTION reduction .
Check against redundacy of the tests. Wheter they kill the same mutants.

Fig. 1. Scheme of our approach.

Our algorithm is divided into several phases that are graphically represented in Figure 1. These phases are:

1. *Generation*. First, the community is generated. A specified number of populations are stored into the community. Second, inhabitants for each population are introduced into the different populations. Each inhabitant holds in its DNA a sequence of inputs that could be applied as a test. Third, the specification is introduced and a set of mutants is generated by applying the two mutation operators.
2. *Evolution*. In the evolution phase each inhabitant tries to kill mutants from a reduced set of candidates. The fitness function is as simple as the percentage of mutants killed from the set. These populations evolve as every other GA during a number of generations, to try and find a solution. The best individuals of each population are selected and saved for further use.
3. *Specialization*. In this phase the already evolved population is modified by inserting one specific faulty IUT into each of the populations, and making the GA try to find suitable individuals that will find the mistake present in the mutation. In this case the fitness function is thought to find one mistake in the IUT with the shortest DNA sequence. The individuals with the shortest test sequence while detecting the fault in the IUt of each population are selected and saved for further use.
4. *Selection*. We sort the inhabitants generated after the previous phases to get the ones that kill more mutants (trying to shorten the overall testing sequence length). More precisely, we initially select the ones that are needed to kill the mutants that did not die when applying the first test, and we repeat the process. Thus, by eliminating redundant tests we have to find a tradeoff between killing the highest number of mutants and getting the shortest sequence. In our framework, we eliminate every test that kills less than 1% of the mutants.

4 Application of GAMuT: Results and Comparisons

We have developed a tool to implement our methodology. We have used Java Technology (JDK 1.6) and the Netbeans software, and made usage of the MVC architecture to enable ease of maintenance, and use session facade, singleton and light-weight design patterns. FSMs are implemented as a set of adjacency matrices, one for each input; instead of having 1 to denote that a connection exists, the corresponding output is specified. The singleton community class holds the objects that must be available at all times. There is a class for each GA object, 3 heuristics that implement a common interface and a couple of auxiliary classes, like matrix and errors.

We have worked with a specification having 50 states. The corresponding FSM has been randomly generated to avoid that its specific structure produces any bias in the application of our methodology. Our set of inputs is $I = \{a, b\}$ and the set of outputs is $O = \{1, 2\}$, but bigger sets could be considered without affecting the results. From the specification we derive 500 mutants, from which 100 are chosen as the learning set. The used specification in the experiment reported in this paper is shown in Figure 2. The number of populations in the community is 30.The size of each population is of 50 inhabitants. The maximal number of generations allowed for the phase of evolution is 200. The maximal number of generations for specialization is set to 200.

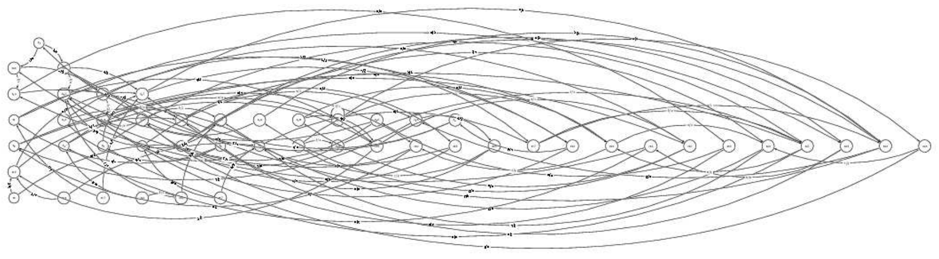

Fig. 2. Example of specification.

DNA are sequences $< i_1 \; i_j \; \ldots \; i_n >$ of inputs. We mutate the length of the DNA sequence in a 33% of the cases, randomly deciding whether to enlarge or to shorten it. We mutate the value of various genes simultaneously, being the number of mutations randomly chosen, but proportionally to the inverse of its fitness function, that is, $n = \frac{random[1, n_g/3]}{f}$, being n the number of genes modified, n_g the total number of genes, and f the value given by the fitness function.

There are two fitness functions: One for the *evolution* phase and another one for the *specialization* phase.

1. Fitness for evolution: $f = \frac{\mid mutantskilled \mid \cdot 100}{\mid mutants \mid}$ is the percentage of mutant implementations, for which an inhabitant finds an error.
2. Fitness for specialization: $f = e - l \cdot \alpha$, where e is equal to 1 if an error is found, and l is the length of the genetic sequence starting to count from the point where the error is found; otherwise, starting to count from the beginning of the sequence, and $0 \leq \alpha \leq 1$ is its weight. This heuristic tries to approximately find the shortest test sequence that detects an error in the IUT that we are checking.

Mating is done through selection with the *roulette wheel* selection technique and it uses a single point *crossover* for DNA reconversion.

In order to check convergence in the *evolution* step, we take into account the fittest individuals from each population, and consider them as a test suite, that is applied to the number of learning IUTs. If the total number of killed mutants is over 85%, then we allow the program to continue towards its next step. For the *specialization* phase we set finding 75% of the errors as a good number to stop the algorithm and go to the selection process. This number does not represent the mutants that the set will kill, but how many of the test found an error in a specific IUT. Obviously, these values can be modified to find fitter tests, but our experience shows that these limits behave good enough. If convergence is not reached before the total number of generations allowed, then the system stops the process and continue with its next step.

In order to compare our methodology we have implemented a random testing tool. Random testing is a technique argued to be as valid as any other testing technique (see [7,9,6]). As we can see in Figure 3, the test suite resulting after selection is the shortest one and detects up to 94.6% of mutants as faulty. Compared to the randomly generated test suite, it outperforms it both by number of mutants killed and by having a shorter test sequence. Even though to produce our test suite takes more time than a random

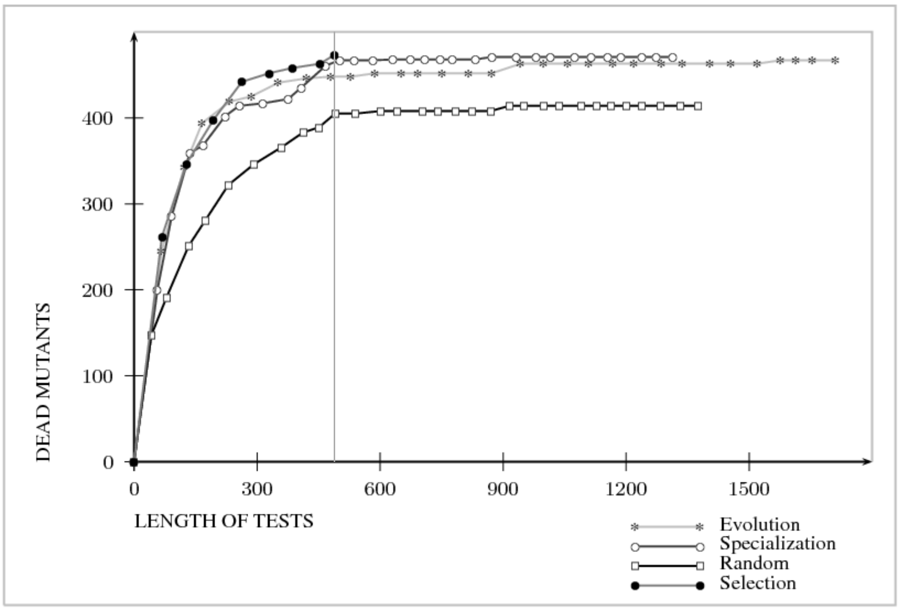

Fig. 3. Killed mutants/length of test sequence.

generation, the extra performance of our approach to find errors, even though not very significant, is worth the additional computations. Actually, the time spent in testing is a valuable asset. Thus, we have to minimize the number of applied tests (in other words, the length of the test suite) even if we have to make additional *off-line* computations, that are very cheap when compared to the cost of testing. Actually, in order to test a system we usually have to stop it. Thus, the more time the system is halted, the most expensive the testing process is, since the system cannot be producing what it is supposed to produce. In the case of the evolution and specialization test suites, they are between 1% and 2% below in test coverage (i.e. in proportion of killed mutants) because we have chosen the best tests of both test suites, and they are a lot longer to apply due to their high redundancy on the kind of errors found. This is even so if we eliminate, from the selected tests, those that kill under 1% of the mutants, because the time to apply those tests, compared to the benefits of the number of mutants detected, makes it not worthy to retain them in the final test suite.

5 Conclusions and future work

The results after experimentation with the implementation of GAMuT have led us to claim that the existence of these different phases is crucial. GAMuT combines the *evolution* as a massive learning process (thanks to mutation testing techniques), in which we try to reach a maximum, and a more local search, *specialization*, that creates elements in a more sophisticated way by creating a variety of viewpoints, seen as genetic diversity, that confronts the same problem. It is also important to remark the order in which the phases are applied, because this modifies the starting point of the population for the second GA step (*specialization*), making it a good starting point to be able to locate an optimum. Furthermore, *selection* is crucial as well because we need to get rid of the redundancy that the overlapping solutions offers, a normal consequence of the stochastic transformations.

We believe that this approach is feasible and since it tends to create a close-to-complete set of tests, with a short amount of testing time, it is suitable for its application on a number of systems, once it is developed for other languages. Due to this fact, one line of future work will be to study how to codify more languages into the DNA sequences and any derivation that will cause this on the rest of the elements of the system, like the mutation operators and the fitness functions.

Another idea for future development is to provide the GA algorithm with memory for its inhabitants. Then, they could foresee the best mutation rates and the ones that are leading them to a better heuristic value, maybe through the use of a neural network or simply by probabilistic methods. In this way we can reach convergence faster.

Acknowledgements: We would like to thank the anonymous reviewers of this paper for the careful reading and their suggestions.

References

1. J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *27th Int. Conf. on Software Engineering, ICSE'05*, pages 402–411. ACM Press, 2005.
2. K. Derderian, R.M. Hierons, M. Harman, and Q. Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. *Computer Journal*, 49(3):331–344, 2006.
3. S.C.P.F. Fabbri, M.E. Delamaro, J.C. Maldonado, and P.C. Masiero. Mutation analysis testing for finite state machines. In *5th IEEE Int. Symposium on Software Reliability Engineering, ISSRE'94*, pages 220–229. IEEE Computer Society Press, 1994.
4. D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In *4th IEEE Int. Workshop on Source Code Analysis and Manipulation, SCAM'04*, pages 65–74. IEEE Computer Society Press, 2004.
5. Q. Guo, R. M. Hierons, M. Harman, and K. Derderian. Computing unique input/ouput sequences using genetic algorithms. In *3rd Int. Workshop on Formal Approaches to Software Testing, FATES'03, LNCS 2931*, pages 169–184. Springer, 2004.
6. D. Hamlet. When only random testing will do. In *1st International Workshop on Random Testing*, pages 1–9. ACM Press, 2006.
7. R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
8. T. Sugeta, J.C. Maldonado, and W.E. Wong. Mutation testing applied to validate SDL specifications. In *16th IFIP Int. Conf. on Testing of Communicating Systems,TestCom'04, LNCS 2978*, pages 193–208. Springer, 2004.
9. H. Zhu, P.A.V Hall, and J.H.R. May. Software unit test coverage and adequacy. *ACM Computing Surverys*, 29(4):366–427, 1997.