ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **11**, Number 3, 2008, 209–227

Genetic Model based Testing: a Framework and a Case Study

F. IPATE, R. LEFTICARU

Department of Computer Science and Mathematics
University of Piteşti, Romania

E-mail: florentin.ipate@ifsoft.ro,
raluca.lefticaru@gmail.com

Abstract. The application of metaheuristic search techniques in test data generation has been extensively investigated in recent years. Most studies, however, have concentrated on the application of such techniques in structural testing. The use of search-based techniques in functional testing is less frequent, the main cause being the implicit nature of the specification. On the other hand, such techniques could be employed in functional test generation if an explicit, graph-based, model, that describes the algorithm used to produce the required results, existed. However, the process of creating and validating such a model is usually a highly-specialized and time consuming task, which quite often cannot be economically justified in the case of non-safety-critical applications. In this paper we propose a framework for genetic model based testing. Under this framework, a graph-based model of the system under test is built using a genetic algorithm. Test data is then derived from the resulting model using (possibly) metaheuristic search techniques to provide the desired level of coverage. The approach is illustrated with a case study: an array sorting program.

Key words: evolutionary testing, genetic algorithms, model based testing, genetic programming.

1. Introduction

The application of metaheuristic search techniques in test data generation has been extensively investigated in recent years [21]. Most studies, however, have concentrated on the application of such techniques in structural (program-based or white-box)

testing [8, 17, 25, 27, 30]. In structural testing, the program is represented as a directed graph, in which each node corresponds to a statement or a sequence of statements and each edge (branch) to a transfer of control between two nodes. Search-based techniques are then used to generate test data to cover the desired graph elements (nodes, branches or paths).

The application of search-based techniques in functional (specification-based or black-box) testing is less frequent, the main cause being the implicit nature of the specification. Typically, metaheuristic search techniques have been used to derive test data from specifications in the form of a set of pre-condition/ post-condition blocks, written in a language such as Z, in which the pre-condition defines valid inputs and the post-condition defines the output. Each such pre-condition can be considered to define a "path" in the specification and search-based techniques can then be used to generate data to exercise each path [14]. Furthermore, the work of Tracey et al. [29, 28, 31] extends this idea by also using the post-condition of each block to validate the output produced along each path. Consequently, a failure is found when an input situation is discovered that satisfies the pre-condition, but for which the outputs violate the post-condition. An objective function of the form pre-condition $\wedge\neg$ post-condition, that measures the "closeness" of the test data to uncovering such a situation, is defined and metaheuristic search techniques are employed to seek failures in the implementation.

Specifications of the type discussed above can inherently be translated into programs composed of "if-else" statements and so the paths defined in the specification can be mapped onto paths in the actual program implementation. However, software functionality is rarely this simple and quite often there is no straightforward mapping between the (possibly formal) specification and the actual implementation. The specification is usually just a description of what a system does, without giving the details of how this is achieved. Consider the case of an array sorting, for example. A specification will just state that the system outputs an array with identical elements as the input array, but ordered from the smallest to the largest (for illustration, a Z "sort" schema is given in Fig. 1 – the names of the predicates used are self-explanatory and so no further detail is provided). The specification does not say anything about what strategy is to be used to implement this functionality (e.g. bubble sort, quick sort, etc.), so, clearly, there is no direct mapping between it and the resulting implementation. Consequently, if the aforementioned test generation techniques are to be used in this case, the specification will have to first undergo a transformation that will allow paths and their corresponding pre-conditions and post-conditions to be clearly identified.

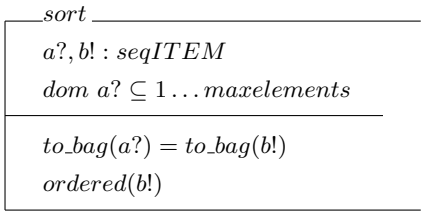

Fig. 1. Z schema for array sorting.

An apparently simple solution to this problem would be to re-write the specification as a set of pre-condition/ post-condition blocks to which the aforementioned techniques can be directly applied. However, for even fairly simple specifications, this approach may yield an extremely large number of such blocks (e.g. the specification of a program that sorts an array of n elements would have $n!$ blocks), so quite often this approach is impractical.

A more reasonable alternative would be to replace the original, implicit, specification with an explicit, graph-based, model that will not only describe what the system is supposed to do but also how it does it, i.e. the algorithm used to produce the required results. Such an explicit model can be produced using state-based languages, such as Statecharts [10] or stream X-machines [11]. A set of test paths can be derived from the model using finite state machine [26] or stream X-machine [3, 12] based methods and metaheuristic search techniques can then be used to generate test data to drive these paths. For instance, genetic algorithms are used in [20] to derive test data from a statechart model of a class and the reported experimental results are encouraging. The main problem with this approach is that it requires the construction of a formal model of the system under test, which may be as complex as the program itself. The process of creating and validating such a model (proving that it satisfies the requirements) is usually a highly-specialized and time consuming task, which quite often cannot be economically justified in the case of non-safety-critical applications.

In this paper we propose an approach whereby an (approximate) graph-based model of the system under test is built using a genetic algorithm. The model is then used as basis for test generation: a set of paths are derived from the model and then these are translated into actual test data using (possibly) metaheuristic search techniques. While, obviously, an exact model of the system is desirable, a close enough approximation is often considerably less computationally intensive to produce and may still provide a reasonable basis for test generation. The approach is illustrated on an array sorting program and experimental results that support the method are reported.

The paper is structured as follows. Section 2 provides a brief overview of genetic algorithms. Section 3 defines the modelling formalism used, called flowchart-machine (F-machine). The generation of F-machine models using genetic algorithms is then discussed in section 4, while section 5 addresses the test derivation problem. Experimental results are provided in section 6 and the next section provides a brief comparison to related work. Finally, conclusions are drawn in section 8.

2. Genetic algorithms

Genetic algorithms (GAs) [33, 23] are a 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 taken from [21] 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 minimising 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.

```
Randomly generate or seed initial population P
Repeat
   Evaluate fitness of each individual in P
   Select parents from P according to selection mechanism
   Recombine parents to form new offspring
   Mutate P'
   Construct new population P' from parents and offspring
   P ← P'
Until Stopping Condition Reached
```

Fig. 2. Genetic Algorithm.

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 [23].

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 probability p_m, usually very small [23]. This operator is responsible for introducing variation in the population.

3. Flowchart-machine

The modelling formalism used in this paper will be called a *flowchart-machine* (abbreviated *F-machine*) and is essentially a formalization of the well-known flowchart, widely used for describing algorithms and processes. Such a model can be easily encoded into a sequence of integers (the chromosome representation of an F-machine) as detailed later in the paper.

Definition 3.1. A *flowchart-machine* (*F-machine*) is a tuple $Z = (In, Out, Q, M, P, \Phi, q_0, q_f, d, h, input, output)$, where:

- In is the (possibly infinite) input set.
- Out is the (possibly infinite) output set.
- Q is the finite set of non-final *states*.
- M is a (possibly infinite) set called *memory*. In our applications, the memory will have the form $M = V_1 \times \ldots \times V_n$, $n \geq 1$, where V_i denotes the domain of some program variable, $1 \leq i \leq n$.
- P is a finite set of distinct *predicates* on M. P may contain the "true" predicate, that holds for all values of M.
- Φ is a finite set of distinct *processing functions* of type $M \longrightarrow M$. Φ may contain the identity function id, used when no actual processing takes place.
- $q_0 \in Q$ is the *initial state*.
- $q_f \notin Q$ is the *final state*.
- d is a function of type $Q \longrightarrow P$, that assigns a predicate to each non-final state. If the predicate associated with the state q is *true* then q is called a *sequence state*, otherwise q is called a *decision state*.
- h is a function of type $Q \times B \longrightarrow (Q \cup \{q_f\}) \times \Phi$, where $B = \{T, F\}$ is the Boolean set. $h(q, T)$ represents the next state and the corresponding processing function for the case in which $d(q)$ is true, whereas $h(q, F)$ represents the next state and the processing function for the case in which $d(q)$ is false. When q is a sequence state, the "false" branch is immaterial, however the definition is given in this form for the sake of uniformity.
- $input : In \longrightarrow M$ is the *input function* that maps an input into an initial memory value.
- $output : M \longrightarrow Out$ is the *output function* that maps a final memory value into an output.

For illustration, consider the graphical representation of an F-machine for an array sorting (using a bubble sort like approach) given in Fig. 3. The predicate associated with each state is given inside the circle representing the state. Each transition between states is labelled by a pair b/ϕ, where $b = T$ for a "true" transition and $b = F$ for a "false" transition and ϕ denotes the processing function associated with the transition. The memory will have to hold the n values to be sorted as well as the two indexes i and j, so $M = \mathbb{R}^n \times \mathbb{N}^2$, where $\mathbb{R}$ is the set of reals and $\mathbb{N}$ the set of positive integers. $In = \mathbb{R}^n$ and $Out = \mathbb{R}^n$ will hold the original and sorted array, respectively, so the input and output function will be defined by: $input(a[1], \ldots, a[n]) = (a[1], \ldots, a[n], 1, 1)$ and $output(a[1], \ldots, a[n], i, j) = (a[1], \ldots, a[n])$, $i, j \in \mathbb{N}$. In Fig. 3 $swap(i, j)$ exchanges the values of the ith and jth components of the memory, $inc(i)$ increases the value of the integer i by 1, while $init(i)$ sets the value of i to 1.

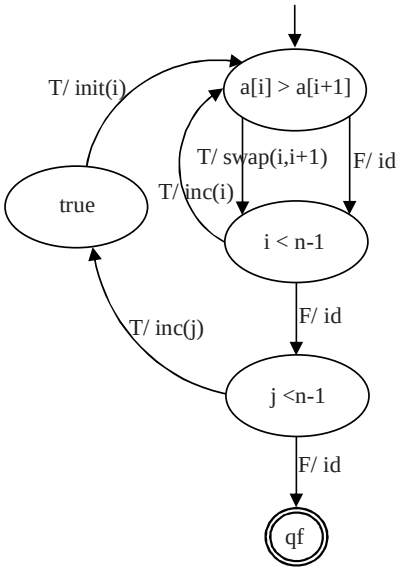

Fig. 3. F-machine model of an array sorting.

The computation of an F-machine takes the form of the traversal of all paths (sequences of arcs) from the initial state to the final state and the application, in turn, of the arc labels (the processing functions) to the initial memory value. The correspondence between the input sequence applied to the machine and the output produced gives rise to the (partial) function computed by the machine.

Definition 3.2. Given an F-machine Z, the *(partial) function computed* by Z, $[Z] : In \longrightarrow Out$, is defined by: $[Z](in) = out$ if there exist processing functions $\phi_1, \ldots, \phi_n$, $n \geq 1$, such that the following hold simultaneously:

- there is a path from the initial state q_0 to the final state q_f labelled by $\phi_1, \ldots, \phi_n$, i.e. there exist non-final states $q_1, \ldots, q_{n-1}$ such that for all i, $0 \leq i \leq n-1$, either $h(q_i, T) = q_{i+1} \wedge d(q_i)$ or $h(q_i, F) = q_{i+1} \wedge \neg\, d(q_i)$, where $q_n = q_f$;

- the sequence of processing functions $\phi_1, \ldots, \phi_n$ transforms m into m', i.e. there exist memory values $m_0, \ldots, m_n$ such that $input(in) = m_0$ and $output(m_n) = out$ and for all i, $0 \leq i \leq n-1$, $\phi_i(m_i) = m_{i+1}$.

It can be observed that, due to the deterministic nature of the F-machine defined above, $[Z]$ is a (partial) function rather than a relation. Also note that $[Z]$ is partial if there is some initial memory value for which M loops indefinitely.

4. F-machine model generation

A genetic algorithm will be used to produce an F-machine model of the system under test. Therefore, the basic approach is to generate a random population of chromosomes (representations of F-machine models of the system under test), evaluate their fitness, perform various genetic operations on them based in some way on their fitness and then re-evaluate the fitness of the newly produced generation. The process continues until a satisfactory model has been found or a maximum number of generations has been produced.

The algorithm may not attempt to construct a precise model of the system. A good approximation may be much less computationally intensive to produce and may still be sufficient for testing purposes. In the case of an array sorting, for example, it should be sufficient to find a model that sorts arrays of n elements, with n fixed and sufficiently large to illustrate the underlying ideas of the algorithm instead of a general model, that will sort arrays of *any size* (up to a given maximum length). For more complex systems, many approximate models, which cover different aspects of the functionality, may be needed.

4.1. Defining the primitives

In order to apply a genetic algorithm to a problem, each candidate solution must be represented in such a way that it can evolve by applying genetic operations. In the case of an F-model, this is achieved by defining a set of suitable primitives, that represent the set of predicates P and processing functions Φ that any model of the system may use. In general, the set of primitives required for the construction of a model, for a particular system, can be chosen from the operations that would normally appear in the implementation, as presented in what follows.

Consider again the array sorting example. If the array is assumed to always have a fixed number of elements, $n \geq 2$, then it will be sufficient to have $M = \mathbb{R}^n$, *input* and *output* the identity function and the following primitives:

- P: the set of predicates $greater(i, j)$, $1 \leq i < j \leq n$, defined by $greater(i, j) = a[i] > a[j]$, where $a[i]$ denotes the value of the ith component of the memory. If the model may also contain sequential states, then $true \in P$.
- Φ: the set of functions $swap(i, j)$, $1 \leq i < j \leq n$, plus the identity function id for when no processing takes place.

The state-transition diagram of a *correct* model for $n = 4$ is as represented in Fig. 4.

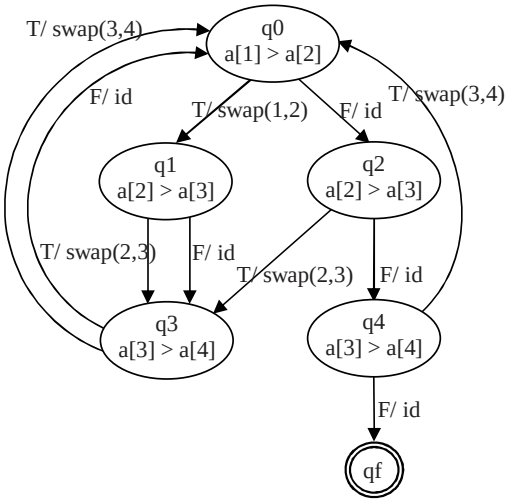

Fig. 4. F-machine model for $n = 4$.

4.2. Encoding the model population

Besides defining the model primitives, we need to establish a reasonable upper bound on the states of the candidate models. This entails assessing the scale of the correct model of the system under test and, along with the definition of the model primitives, requires some knowledge of the way the system has been implemented. As shown by our experiments (Table 3), a too tight an upper bound (i.e. very close to the minimum possible number of states of a correct model) will reduce the degree of freedom of the candidate models and hence increase the number of evolutions required to produce a solution. Conversely, a too large an upperbound will produce unjustifiably large solutions, thus increasing the duration of an evolution and unnecessarily complicating the test generation process (see section 5).

An F-machine with (at most) k non-final states can be represented as a sequence of $5 \cdot k$ integers, by encoding the information about each state into the 5 elements as follows:

- 1 : the predicate associated with the state.
- 2 and 3: the next-state and the corresponding processing function for when the predicate associated with the state is true.
- 4 and 5: the next-state and the corresponding processing function for when the predicate associated with the state is false.

Obviously, this encoding corresponds to the worst case scenario, in which the space of candidate models corresponds to the set of *all* F-machines of the given size that can be constructed from the set of available primitives. Naturally, additional

knowledge about the system will normally be available and this will enable a more precise delimitation of the solution space and hence a more efficient encoding. For the array sorting example, one can naturally assume that a $swap(i, j)$ function will always be attached to the true branch of a $greater(i, j)$ predicate, whereas no processing (i.e. the identity function) will be associated with the false branch. In this case, only 3 integers will be sufficient to encode the information about a state.

4.3. Fitness function

The fitness function will measure how far the output produced by an F-machine model is from the expected output. In the case of an array sorting, the fitness function will measure the "disorder" of the output array. Given two elements of the array, $a[i]$ and $a[j]$, $1 \leq i < j \leq n$, the disorder of the pair $(a[i], a[j])$ can be measured by Korel's objective function: $d_{ij} = 0$ if $a[i] \leq a[j]$ and $a[i] - a[j]$ otherwise. In our experiments, we used a normalized version of this function: $f_{ij} = 1 - 1.001^{-d_{ij}}$. Thus, the fitness function for an array $a[1], \ldots, a[n]$ was $f(a[1], \ldots, a[n]) = \sum_{1 \leq i < j \leq n} f_{ij}$.

The genetic algorithm is provided with a set of test data to be used for fitness evaluation. A "global" fitness of each candidate model will be then calculated as the average of the fitness values obtained for each individual test value.

An important aspect to consider is the selection of the test data. Often, the input domain is very large or even infinite and so not all possible values can be used for fitness evaluation. In such cases, a set (typically randomly generated) of sufficient size to provide a good representation of the problem will be needed to train the algorithm. In the case of the sorting algorithm, an extreme situation would be to present all candidate models with only one sequence of numbers for the entire run. In this case, the algorithm will evolve quickly towards a model which will correctly order the given sequence, but it is extremely unlikely that it will constitute a correct model of a general sorting algorithm. Interestingly, our experiments have shown that changing the test data periodically (e.g. each generation) will increase the performance of the genetic algorithm only if the test set is relatively small (e.g. 5 arrays for a model that sorts arrays of 4 elements); for sufficiently large test sets, it may have an adverse effect.

5. Test data generation

The model thus found is then used as basis for test generation: test data is selected to achieve the required degree of graph coverage. Obviously, the selection procedure will depend on the level of coverage sought. For example, a set of paths that cover all branches of the F-machine graph can be constructed by devising a test tree in a breadth-first manner as in [5]. A node in the tree is a leaf if there are no edges emerging from it (e.g. the final state of the F-machine) or if its label is the same as that of a non-leaf node that has already been encountered. The tree obtained for the array sorting model example is as represented in Fig. 5. The leaf nodes are in bold.

Then branch coverage is achieved by selecting all total paths in the tree (i.e. from the initial node to each of the leaf nodes). Alternatively, the tree could be constructed in a depth-first fashion as in [2].

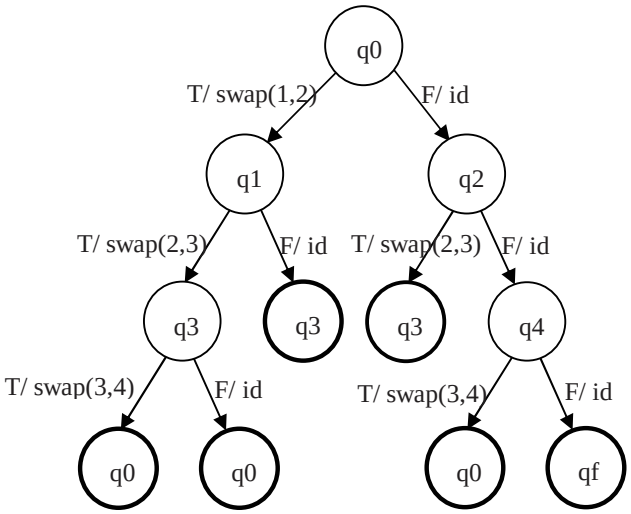

Fig. 5. Test tree for array sorting.

Once the paths have been selected, we need to find appropriate input values to exercise the selected paths. A commonly used technique for this purpose is symbolic execution [15, 6]. Rather than running the program on actual input values, this technique derives a set of constraints in terms of the input variables which describe the conditions necessary for the traversal of a given path. For example, the leftmost path $< q_0 \xrightarrow{T} q_1 \xrightarrow{T} q_3 \xrightarrow{T} q_0 >$ will produce the following constraints: $a[1] > a[2]$, $a[1] > a[3]$, $a[1] > a[4]$. Constraint satisfaction problems are in general NP-complete [6]. However, linear programming techniques can be applied if the constraints are linear [6]. If this is not the case, constraint satisfaction based test generation techniques, such as domain reduction [7] or dynamic domain reduction [24] can be used. However, these static test generation techniques typically suffer from problems due to computed storage locations and loops. Alternatively, metaheuristic search techniques, such as genetic algorithms or simulated annealing [30, 28] can be applied to dynamically generate test data that meet the desired level of coverage [25, 31, 32].

Furthermore, the test generation strategy can be extended to check the conformance of the implementation to the F-machine model by applying test data to the implementation and then validating the output against the model as in the work of Tracey et al. [29, 28, 31]. Suppose we have determined a set of paths from the initial state of the F-machine to its final state that achieve the desired level of graph coverage. The traversal of each such path will yield the corresponding pre-condition (in terms of input variables) and post-condition (in terms of input and output variables) for the path. For example the path $< q_0 \xrightarrow{T} q_1 \xrightarrow{T}$

$q_3 \xrightarrow{T} q_0 \xrightarrow{F} q_2 \xrightarrow{F} q_4 \xrightarrow{F} q_f >$ from our F-machine example will produce the precondition $(a[1] > a[2]) \wedge (a[1] > a[3]) \wedge (a[1] > a[4]) \wedge (a[2] \leq a[3]) \wedge (a[3] \leq a[4]) \wedge (a[4] \leq a[1])$ and the post-condition $(b[1] = a[2]) \wedge (b[2] = a[3]) \wedge (b[3] = a[4]) \wedge (b[4] = a[1])$, where b denotes the output array (the evolution of memory variables is described in Table 1). An objective function of the form *pre-condition* $\wedge\neg$ *post-condition* is then defined and metaheuristic search techniques are used to seek faults in the implementation. To improve the power of the search, the pre-condition and negated post-condition are first converted to Disjunctive Normal Form (DNF). For example, $A \wedge B \wedge \neg(C \wedge D)$ will be decomposed into two disjuncts, $A \wedge B \wedge \neg C$ and $A \wedge B \wedge \neg D$ and so finding a solution to the original expression will be reduced to finding a solution to any of the two disjuncts. Consequently, each disjunct will be targeted in a separate search attempt. The fitness function is then calculated according to the rules in Table 4, where the value K, $K > 0$, refers to a constant which is always added if the term is not true.

Table 1. Variable evolution for the transition path in the first column

Transition	Predicate	New array values	Precondition
$q_0 \xrightarrow{T} q_1$	$a_1 > a_2$	$[a_2, a_1, a_3, a_4]$	$a_1 > a_2$
$q_1 \xrightarrow{T} q_3$	$a_2 > a_3$	$[a_2, a_3, a_1, a_4]$	$a_1 > a_3$
$q_3 \xrightarrow{T} q_0$	$a_3 > a_4$	$[a_2, a_3, a_4, a_1]$	$a_1 > a_4$
$q_0 \xrightarrow{F} q_2$	$a_1 \leq a_2$	$[a_2, a_3, a_4, a_1]$	$a_2 \leq a_3$
$q_2 \xrightarrow{F} q_4$	$a_2 \leq a_3$	$[a_2, a_3, a_4, a_1]$	$a_3 \leq a_4$
$q_4 \xrightarrow{F} q_f$	$a_3 \leq a_4$	$[a_2, a_3, a_4, a_1]$	$a_4 \leq a_1$

Thus, test generation can be regarded as a two stage process. In the first stage, test data is derived from the F-machine model to achieve the desired level of coverage. Once these initial values, and implicitly the corresponding paths through the model, have been selected, test data is derived from the implementation under test (second stage). In the second stage, for each chosen path, test values are selected according to a fitness function that indicates how close to producing a fault on the given path the implementation is.

6. Experimental results

In the first stage of our experiments, we investigated the generation of F-machine models for sorting an array of fixed length $n, n > 3$, using genetic algorithms.

Several encodings for the F-machine were tried, by varying the primitives and the number of elements used for encoding. Three sets of predicates were used: (a) $greater(i, j)$, with $1 \leq i \neq j \leq n$, (b) $greater(i, j)$, with $1 \leq i < j \leq n$ and (c) $greater(i, i + 1)$, with $1 \leq i < n$. The F-machine was encoded (1) on $5 \cdot k$ integers

(the general case, as described in subsection 4.2) and (2) on $3 \cdot k$ integers. The second encoding always attached the *swap* function to the true branch for $i < j$. As expected, the second type of encoding, combined with the predicate set (c) produced the fastest convergence of the genetic algorithm. The results given further in this paper (Tables 2 and 3) are those produced in this case. For comparison, the success rate for combinations (a)(1), (b)(1), (c)(1), (a)(2), (b)(2) and (c)(2) (for $n = 4$, $k = 5$ and 24 learning arrays permutations - see below for details) was 59%, 67%, 90%, 93%, 97% and 98%, respectively, and the average generation number approximately 69, 58, 31, 38, 27 and 16, respectively.

Table 2. Genetic algorithm results for evolving F-machines, $n = 4$, $k = 5$

Samples used to evolve the F-machines	Correct F-machines	Failures	First gen.	Avg. gen.	Difference
5 Fixed arrays	90.20%	9.80%	10.09	26.91	16.82
5 Random arrays	96.50%	3.50%	7.39	18.60	11.21
10 Fixed arrays	93.50%	6.50%	12.35	20.03	7.68
10 Random arrays	96.00%	4.00%	10.27	18.27	8.00
5 Fixed arrays + 5 random arrays	95.20%	4.80%	10.87	18.29	7.42
16 Fixed arrays	97.50%	2.50%	13.90	16.20	2.30
16 Random arrays	95.80%	4.20%	11.04	17.06	6.02
16 Binary arrays	99.70%	0.30%	11.33	11.75	0.42
6 Fixed arrays + 10 random arrays	96.40%	3.60%	12.41	16.73	4.32
24 Fixed arrays	97.70%	2.30%	14.82	15.65	0.83
24 Random arrays	95.30%	4.70%	13.72	17.40	3.68
24 Permutations	98.30%	1.70%	15.77	15.77	0.00
10 Fixed arrays + 14 random arrays	96.30%	3.70%	14.09	16.21	2.12

The algorithms were implemented in Java, using JGAP (Java Genetics Algorithm Package) [13]. An elitist genetic algorithm was used, with the following default parameter values: a population size of 20 individuals and a maximum allowed number of evolutions equal to 100. The JGAP operators used were *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 *MutationOperator*, with a default 1/15 mutation rate. A heuristic real value crossover inspired from [22] was used for recombination. This uses the values of the objective function for determining the direction of the search. Given parents $x = (x_1, \ldots, x_n)$, $y = (y_1, ..., y_n)$, x fitter than y, this approach will generate one offspring $z = (z_1, ..., z_n)$ with $z_i = \alpha \cdot (x_i - y_i) + x_i$, $\alpha \in (0, 1)$. Instead of an $\alpha \in (0, 1)$ randomly obtained, as in [22], we considered a fixed $\alpha = 0.2$, for which better performance was obtained.

Table 3. Genetic algorithm results for evolving F-machines

n : array length	k : state number	Correct F-machines	First gen.	Avg. gen.
4	3	64.40%	35.81	50.81
4	4	89.80%	18.89	25.06
4	5	95.90%	13.39	17.22
4	6	98.70%	11.06	12.56
4	7	98.90%	10.23	11.64
4	8	99.33%	8.77	10.09
5	4	47.67%	49.39	69.05
5	5	71.33%	31.92	50.18
5	6	83.33%	25.83	38.64
5	7	88.33%	23.46	34.22
5	8	92.00%	22.68	31.17
5	9	95.33%	20.51	27.92
5	10	97.33%	19.77	26.41
6	5	32.67%	61.90	82.93
6	6	49.33%	51.99	75.63
6	7	62.00%	44.11	68.12
6	8	73.67%	38.35	58.26
6	9	75.00%	35.36	55.65
6	10	76.67%	35.48	55.39
6	11	81.67%	33.89	51.68
6	12	83.00%	30.00	50.62

The fitness function used to evaluate the F-machine adequacy measured the disorder of the output array obtained after the sorting. The formula was given in subsection 4.3[1]. Furthermore, the fitness of an F-machine, as calculated by the algorithm, may be 0 but the machine may only sort correctly the provided samples. For this reason, an additional "test for generality" was performed whenever an F-machine with fitness 0 was found. This involved evaluating the machine fitness on 100 randomly generated arrays, the set of all permutations of $\{1, 2, \ldots, n\}$ and the set of all binary arrays - the sequences of length n of 0s and 1s.

Table 2 shows the results obtained for evolving F-machine models with $k = 5$ states for sorting arrays of length $n = 4$. The second column in the table shows the percentage of correct F-machine models evolved, while the third column records the percentage of unsuccessful runs (i.e. when the genetic algorithm could not evolve an F-machine with $fitness = 0$ that passes the test for generality in the maximum allowed number of evolutions, 100). The fourth column indicates the average of the first generation when an individual with fitness 0 was found (usually, this failed the test for generality), while the fifth column shows the average number of generations needed to evolve a completely fit individual (which passes the test for generality). The last column is the difference between the 5th and the 4th columns.

[1]The fitness function is assigned the maximum allowed value when the model loops indefinitely on the given input.

The algorithm was trained using several types of "learning" arrays: fixed arrays (chosen at random); variable arrays (random arrays changed each generation); combinations of fixed and variable arrays; binary arrays; the set of all permutations of $\{1, 2, 3, 4\}$. Binary arrays and permutations offer a good sampling of the input domain and are used mainly as terms of comparison.

Our experiments show that changing the (randomly generated) samples periodically (e.g. each generation) will increase the performance of the genetic algorithm only if the sample is relatively small (e.g. $5 - 10$ arrays for $n = 4$). For larger test sets (e.g. 24 arrays for $n = 4$), changing the test data periodically will slightly reduce the number of generations needed to produce a candidate with fitness 0 on the given samples, but the first solution found will typically be more likely to fail the test for generality than the first solution produced in the case of fixed test data. Furthermore, for reasonably large test sets (e.g. 24 arrays for $n = 4$), the algorithm will perform almost as well on random samples as on representative samples, such as permutations, whose selection assumes knowledge of the problem to be solved.

Table 3 summarizes the results obtained for $n \in \{4, 5, 6\}$ and $k \in \{n-1, \ldots, 2 \cdot n\}$ when the samples provided to the machines were 25 variable arrays (random arrays changed each generation). As above, an additional test for generality was performed as well. As expected, an increase in the (maximum) number of states k resulted in a a reduction of the number of evolutions required to produce a solution. On the other hand, a too large k will obviously produce overly complicated and ultimately unusable models.

In the second stage of our experiments, the "correct" F-machine models generated in the first stage were used as basis for test generation to achieve branch coverage. It is worth noting that many of the F-machines models obtained have unreachable states (e.g. state q_1 from Fig. 6[2]) and transitions that cannot be covered (e.g. the two transitions from the unreachable state q_1 and the true branch from q_4). In order to achieve branch coverage, as discussed in section 5, a test tree is constructed first. Then, for each path in the tree (from the initial node to the leaf nodes) the test data is obtained using a genetic algorithm.

Analogously to [20], the fitness of the test data (in our case arrays of length 4) aimed at covering a given path in the tree is calculated using the formula:

$$fitness = approach\_level + normalized\_branch\_level$$

The *approach level* is 0 for individuals which follow the given path, otherwise it is calculated by subtracting one from the number of branches lying between the node from which the individual diverged away from the path and the last node. The *branch level* will compute, for the place where the actual path diverges from the required one, how close is the precondition predicate to being true. The *normalized branch level* can be derived from the guard predicates using the transformations given in Table 4 and then a function that maps every positive value onto $[0, 1]$.

[2]Dashed arrows represent false branches with the identity function, while standard arrows are used for true branches with the swap function.

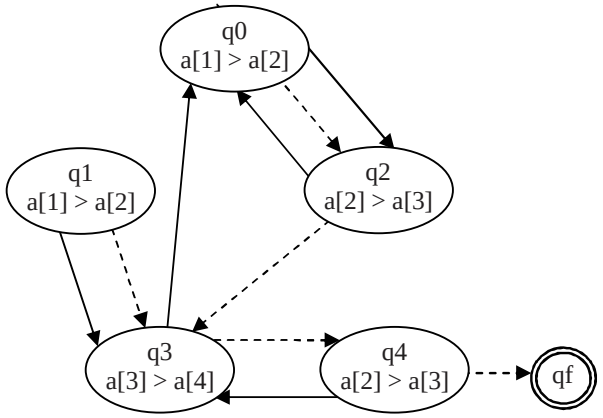

Fig. 6. F-machine model for $n = 4$ automatically generated.

Table 4. Tracey's objective functions

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 $(b - a) + K$
Boolean	if $true$ then 0 else K
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
a xor b	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

The average number of paths in the test tree obtained for $n = 4$ and $k = 5$ was 5.3 (calculated from 1000 executions of the program) so the resulting test data was usually represented as 5 arrays of length $n = 4$. The test data was successfully generated for all the *feasible* paths of the F-machine in just a few generations.

In order to evaluate the effectiveness of our approach, we used the test data generated for $n = 4$, $k = 5$ to build JUnit test suites for a set of 17 different implementations (classical sorts like bubble, quick, insertion, shell, heap, merge or selection sort and also some non-conventional algorithms for sorting arrays of fixed length 4). The cyclomatic complexity of the Java implementations was between 9 and 26; the overall line coverage obtained for the classical algorithms was 100% (rounded), while the branch coverage was 99%, as measured using *jcoverage* Eclipse plugin. To avoid any bias, the same test suite was used against all implementations. The worst results were obtained for one of the non-conventional sorting algorithms (80% and 93%, respectively), but this was mainly due to the large number of paths in the implementation (corresponding to imbricated if statements for all possible cases).

7. Related work

Genetic programming (GP) [1, 18, 19] is a general purpose method for automatically generating a computer program. Concepts from genetic algorithms are used to evolve a population of computer programs according to a fitness landscape determined by the ability of a program to perform a given computational task. GP evolves computer programs represented in memory as tree structures, which can be naturally built in a recursive manner. Every tree node has an operator function and every terminal node has an operand, making mathematical expressions easy to evolve and evaluate. Consequently, traditionally GP favours the use of programming languages that naturally embody tree structures such as Lips or other functional languages. GP is very computationally intensive and so in the 1990s it was mainly used to solve relatively simple problems. More recently, however, improvements in GP technology, growth in CPU power and the parallelization of GP have considerably widened its application area, from hardware design to quantum computing and game playing.

Even though in this paper we advocate the automatic construction of a model of a system using a genetic algorithm, at least two aspects differentiate our approach from GP. Firstly, a model of the system is generated, rather than the program itself. The model may not be at the lowest possible level. On the contrary, using his/her knowledge of the problem to be modelled, the designer (tester) will be expected to use sufficiently high level primitives for the model in order to facilitate a rapid convergence of the genetic algorithm. Furthermore, when an exact model may be too computationally intensive to produce, one or more approximate models may be used as basis for test generation instead. Secondly, the control flow of a program written in a main-stream imperative language is graph-based rather than tree-based. Consequently, a graph-based model would be the natural choice to use in test generation.

Interestingly, the GP paradigm is applied by Kinnear [16] to the task of evolving iterative sorting algorithms. The most concise of the sorts evolved in the runs, which had a total of 42 functions and terminals, would still have to have been extensively hand simplified before could be of any use for the tester, as a realistic model. Furthermore, the next smallest program evolved had 78 functions and operations and, typically, they were in the 150 to 300 range.

A particular subset of GP, called linear genetic programming (LGP) [4], that evolves computer programs as sequences of imperative instructions, also exists. Its proponents claim that, due to its graph-based data flow and techniques for detecting and eliminating non-effective code, it can produce more compact program solutions than the traditional, tree-based, GP. Still, the differences from our approach with regard to its purpose and the level of primitives remain.

8. Conclusions

In this paper we propose a framework for genetic model based testing. Under this framework, an (approximate) graph-based model of the system under test is built using a genetic algorithm and test data is then derived from the resulting model using

(possibly) metaheuristic search techniques to provide the desired level of coverage. The approach is illustrated on an array sorting program.

The process of creating and validating a model for the system under test is usually a costly and time consuming task. Under this approach, this process can be fully automated. Test data can also be automatically derived from the resulting model. The benefit of using this kind of approach, compared to an evolutionary structural testing strategy, is the usage of model post-conditions. In this way the conformance of the implementation with the specification can also be tackled.

Often an exact model may be too computationally intensive to produce. In this case, many approximate models, that cover different functional aspects, may be generated and used for testing purposes.

The approach proposed here could be complemented with traditional functional methods based on domain partitioning. For example, in the case of an array sorting, it would be natural to consider separate test cases for when the array size is 0, 1, $max - 1$, max and $max + 1$, where max denotes the maximum size. On the other hand, when it is not straightforward to describe the system processing in terms of its inputs and outputs, traditional functional techniques may provide only a shallow coverage of the system functionality and, consequently, they may not be sufficient on their own.

Naturally, further case studies and experimental work is needed to better evaluate the merits of the approach and to develop it further. Ultimately, appropriate tools will have to be produced.

References

[1] BANZHAF W., NORDIN P., KELLER R., FRANCONE F., *Genetic programming - an introduction: on the automatic evolution of computer programs and its application*, Morgan Kaufmann, 1998.

[2] BINDER R. V., *Testing object-oriented systems: models, patterns, and tools, Object Technology*, Addison-Wesley, 1999.

[3] BOGDANOV K., HOLCOMBE M., IPATE F., SEED L., VANAK S., *Testing methods for X-machines: a review*, Form. Asp. Comput., **18**, 1, 2006, pp. 3-30.

[4] BRAMEIER M., *On linear genetic programming*, PhD thesis, University of Dortmund, 2004.

[5] CHOW T. S., *Testing software design modeled by finite-state machines*, IEEE Trans. Softw. Eng., **4**, 3, 1978, pp. 178-187.

[6] CLARKE L., *A system to generate test data and symbolically execute programs*, IEEE Trans. Softw. Eng., **2**, 3, 1976, pp. 215-222.

[7] DeMILLO R. A., OFFUTT A. J., *Constraint-based automatic test data generation*, IEEE Trans. Softw. Eng., **17**, 9, 1991, pp. 900-909.

[8] FERGUSON R., KOREL B., *The chaining approach for software test data generation*, ACM Trans. Softw. Eng. Methodol., **5**, 1, 1996, pp. 63-86.

[9] GOLDBERG D. E., DEB K., *A comparative analysis of selection schemes used in genetic algorithms*, in Proceedings of the 1st Foundations of Genetic Algorithms, 1990, pp. 69-93.

[10] HAREL D., POLITI M., *Modeling reactive systems with statecharts: the STATEMATE approach*, McGraw-Hill, New York, 1998.

[11] HOLCOMBE M., IPATE F., *Correct systems: building a business process solution*, Springer Verlag, London, 1998.

[12] IPATE F., *Testing against a non-controllable stream X-machine using state counting*, Theoretical Comput. Sci., **353**, 1–3, 2006, pp. 291-316.

[13] JGAP home page, http://jgap.sourceforge.net/, last accessed July 2008.

[14] JONES B., STHAMER H., EYRES D., *The automatic generation of software test data sets using adaptive search techniques*, in Proceedings of 3rd International Conference on Software Quality Management, 1995, pp. 435-444.

[15] KING J., *Symbolic execution and program testing*, Communications of the ACM, **19**, 7, 1976, pp. 385-394.

[16] KINNEAR K. E., *Generality and difficulty in genetic programming: evolving a sort*, in The Fifth International Conference on Genetic Algorithms, 1993, pp. 287-294.

[17] KOREL B., *Automated software test data generation*, IEEE Trans. Softw. Eng., **16**, 8, 1990, pp. 870-879.

[18] KOZA J. R., *Genetic programming: on the programming of computers by means of natural selection (complex adaptive systems)*, MIT Press, 1992.

[19] KOZA J. R., KEANE M.A., STREETER M.J., MYDLOWEC W., LANZA G., YU J., *Genetic programming IV: routine human-competitive machine intelligence*, Kluwer Academic Publishers, 2003.

[20] LEFTICARU R., IPATE F., *Automatic state-based test generation using genetic algorithms*, in SYNASC 07: Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE Computer Society, 2007, pp. 188-195.

[21] McMINN P., *Search-based software test data generation: a survey*, Softw. Test., Verif. Reliab., **14**, 2, 2004, pp. 105-156.

[22] MICHALEWICZ Z., *Genetic algorithms + data structures = evolution programs (3rd ed.)*, Springer-Verlag, 1996.

[23] MITCHELL M., *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, USA, 1998.

[24] OFFUTT A. J., JIN Z., PAN J., *The dynamic domain reduction procedure for test data generation*, Software - Practice and Experience, **22**, 2, 1999, pp. 167-193.

[25] PARGAS R. P., HARROLD M. J., PECK R., *Test-data generation using genetic algorithms*, Softw. Test., Verif. Reliab., **9**, 4, 1999, pp. 263-282.

[26] SIDHU D. P., LEUNG T.-K., *Formal methods for protocol testing: a detailed study*, IEEE Trans. Softw. Eng., **15**, 4, 1989, pp. 413-426.

[27] TONELLA P., *Evolutionary testing of classes*, in ISSTA 04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, ACM, 2004, pp. 119-128.

[28] TRACEY N., *A search-based automated test-data generation framework for safety-critical software*, PhD thesis, University of York, 2000.

[29] TRACEY N., CLARK J. A., MANDER K., *Automated program flaw finding using simulated annealing*, in ISSTA 98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, 1998, pp. 73-81.

[30] TRACEY N., CLARK J. A., MANDER K., McDERMID J. A., *An automated framework for structural test-data generation*, in ASE 98: Proceedings of the 13th IEEE international conference on Automated software engineering, 1998, pp. 285-288.

[31] TRACEY N., CLARK J. A., McDERMID J. A., MANDER K., *A search-based automated test-data generation framework for safety-critical systems*, Springer-Verlag, New York, 2002, pp. 174-213.

[32] WEGENER J., BARESEL A., STHAMER H., *Evolutionary test environment for automatic structural testing*, Information and Software Technology, **43**, 14, 2001, pp. 841-854.

[33] WHITLEY D., *A genetic algorithm tutorial*, Statistics and Computing, **4**, 1994, pp. 65-85.