International Journal of Software Engineering and Knowledge Engineering
Vol. 12, No. 6 (2002) 691–709

AUTOMATIC TEST DATA GENERATION FOR PROGRAM PATHS USING GENETIC ALGORITHMS*

PAULO MARCOS SIQUEIRA BUENO† and MARIO JINO‡
Department of Computer Engineering and Industrial Automation,
School of Electrical and Computer Engineering,
State University of Campinas (Unicamp),
400 Albert Einstein Avenue, 13083-970, Campinas, SP, Brazil
†*bueno@dca.fee.unicamp.br*
‡*jino@dca.fee.unicamp.br*

A new technique and tool are presented for test data generation for path testing. They are based on the dynamic technique and on a Genetic Algorithm, which evolves a population of input data towards reaching and solving the predicates along the program paths. We improve the performance of test data generation by using past input data to compose the initial population for the search. An experiment was done to assess the performance of the techniques compared to that of random data generation.

Keywords: Software testing; test data generation; dynamic technique; genetic algorithms; case-based reasoning.

1. Introduction

Software development is (still) heavily based on human work. Because of human innate inability to communicate and work perfectly, errors happen and often result in faults in the software. Software testing is a critical activity for software quality in which the software is executed with *test cases* aiming at revealing these faults. It is an expensive activity that can account for 50% of the total cost of software development.

Structural testing criteria use the implementation to determine the components of a program to be exercised by the tester. The task of selecting test data to exercise these components is extremely complex and accounts for a large portion of the overall cost of testing. It requires a careful analysis of the program code, a good deal of mental effort, in addition to knowledge of the underlying concepts of the criteria. The automation of this task is crucial [16, 27] but infeasible in general cases, since the problem is equivalent to the undecidable "halting problem" [6].

Techniques and tools have been developed to bring some degree of automation

*This work was partially supported by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

to test data generation. Generally the techniques are either random [3] or emphasize a test philosophy (functional, structural or fault-based). Structural based test data generators are often classified as: path oriented [18, 11, 14, 6] — a program path is selected and an input data set is generated to execute the path or goal oriented [8, 17, 22, 28, 26] — an input data set is generated to execute a statement irrespective of the path taken (some authors use "goal" with a broader meaning, such as data-flow associations or branches).

This work presents a new technique for path-oriented test data generation for programs. The technique is based on the actual execution of an instrumented version of the program under test. Because of its dynamic nature and a carefully defined instrumentation model, the technique can handle arrays, strings, pointers, loops and complex arithmetic expressions. We use a Genetic Algorithm, a search algorithm based on the mechanisms of evolution and natural genetics [12, 39], to explore the program's input domain, searching for an input data set that executes the desired path. Because of the Genetic Algorithm's ability in dealing with complex restrictions and the accuracy of the fitness function defined, we achieve a high success rate in test data generation. We present a new technique for choosing the initial search point (search region in our case) using "past information" to improve the performance of test data generation. Results are presented from an empirical evaluation done to assess the cost and the effectiveness of test data generation using the proposed techniques.

2. Background

The symbolic execution [4, 6, 30, 15, 7] and the dynamic technique [23, 18, 8, 11, 14] are approaches for the generation of test data in structural and fault-based testing. In symbolic execution a static representation is obtained of the execution conditions for a given path as a function of the program's input variables and a search is made for solutions that satisfy the conditions and cause the path's execution. Because of its static nature this approach places restrictions to the treatment of loops, arrays and pointers. In the dynamic technique actual values are assigned to input variables and the program's execution flow is monitored; if it deviates from the intended one, search or numerical methods are used in the attempt to find input values which make the desired flow to be taken.

The structure of a program P can be represented by a directed graph called Control Flow Graph (CFG) $G = (N, E, s, e)$, where N is a set of nodes, E a set of edges, s the unique entry node, and e the unique exit node. A node is a set of statements that are always executed together, an edge (n_i, n_j) corresponds to a possible control transfer between the nodes n_i and n_j. An edge (n_i, n_j) is called a branch if the last statement of n_i is a selection statement or an iteration statement. A branch predicate can be assigned to a branch.

A path is a node sequence $(n_1, n_2, \ldots, n_k)$, $k \geq 2$ such that an edge exists from n_i to n_{i+1}, $1 \leq i \leq (k-1)$, $n_1 = s$ and $n_k = e$. Moreover, a desired path is a path

we want to execute. The number of nodes on a path is its length.

An input variable x_i of a program P is a variable which appears in an input statement, an input parameter, or a global variable used in P. The type of an input variable can be any of all the different types of programming language. The domain D_{xi} of the input variable x_i is the set of all the values x_i can hold. The input domain D of the program P is the cross product of the n input variables domains $D = D_{x1} \times D_{x2} \times \ldots \times D_{xn}$.

An input data set *id* is a set of values assigned to input variables; *id* represents a single point in the n-dimensional input space D.

A path is feasible if there is an input data set for which the path is traversed during program execution; otherwise, the path is infeasible. The objective of the presented techniques is to find input data sets for which the desired path is executed or to determine the likely unfeasibility of the path, if this is the case.

3. Related Work

Miller and Spooner introduce the concept of dynamic test data generation: using the program itself as an efficient procedure for evaluating the path's constraints and generating test data. In their technique integer parameters for a given program are established by the tester and real values are sought to solve the paths constraints using numerical maximization methods [23].

Korel [18] reduces the problem of finding input data to a sequence of subgoals, each one consisting of satisfying a path predicate. Each subgoal is solved by using local direct search to minimize the value of the error's functions associated to the predicates.

Ferguson and Korel extend this technique by including data dependence information for path selection (the "chaining approach") [8]. They use data flow analysis to identify nodes that must be executed prior to reaching a branch to increase the chance of altering the flow execution as desired.

Tracey *et al.* present a test data generation framework based on an optimization technique called Simulated Annealing. They apply this technique for searching test data for structural testing and generalize the approach for testing specification failures, boundary values, assertion exceptions and component reuse [34, 35]. A technique for automatic generation of data to test exception conditions using a Genetic Algorithm is also presented [36].

Gupta *et al.* [13, 14] suggest a method for path testing by determining a set of commands that influence each predicate and a linear arithmetic representation of the predicates. For linear predicates the linear representations are precise and are used to generate input data. If a predicate on the path is non-linear the arbitrarily chosen input is iteratively refined using a "Unified Numerical Approach (UNA)".

Roper *et al.* [32] present an approach where the population of the Genetic Algorithm is the set of test data sets (each individual is a test data set) and the fitness of an individual (a measure of the individual's quality) corresponds to the

coverage achieved in the program under test. The algorithm evolves the population to achieve a desired level of branch coverage.

Jones *et al.* [17] and Michael *et al.* [22] deal with branch testing. A Genetic Algorithm is applied to minimize the fitness function that measures the distance to the execution of a program branch. It is necessary to discover an input data set reaching the branch; however, the fitness function does not distinguish which input data are closer to reaching it. Thus, branches can be considered only incrementally along the control flow graph, making these approaches unsuitable for selective testing of branches or for testing of more complex structural components.

Michael *et al.* [21] use simulated annealing, genetic algorithm and gradient descendent minimization (implemented in the GADGET system) for generating test data for *condition-decision coverage*. They present experimental results for a complex program (2,000 lines of code). As in [17] and [22] this approach focus on obtaining complete coverage according to some criterion, but is inefficient in exercising a certain feature of the code under test.

Pargas *et al.* apply a Genetic Algorithm to search for test data that exercise program "targets" (statements or branches in the current version) [28]. A fitness function measures the number of executed predicates with regard to the path that reaches the target in the Control-Dependence Graph (CDG). However, the fitness values do not take into account which input data are closer to satisfying each predicate, making the search poorly guided (e.g., in the predicate $if(y == 100.0)$ input data reaching the predicate with $y = 99.9$, receive the same fitness values as input data reaching it with $y = 1.0$, despite the fact that the first one is much "closer" to satisfying the predicate). Since CDG paths are acyclic this approach is not suitable for loop testing or for covering more complex targets (such as def-use pairs that may require loop iterations to be covered).

4. Overview of the Techniques

Initially, the program is instrumented by selectively inserting statements to provide information on the execution useful to drive the data generation. The tester selects and writes into a file the set of paths to be executed based on the desired coverage criteria. Paths can also be automatically selected according to some strategy to cover the structural elements (e.g., as defined in [25, 2, 9, 29]). In both cases it is possible to choose specific paths to selectively exercise structural elements not yet covered by other test sets (e.g., functionally or randomly generated sets) or to cover structural elements having likely been affected by software maintenance. Another possibility is to select sub-paths from the entry node (s) to some goal node gn different from the exit node ($gn \neq e$). In this case the tool searches for input data to force the execution to follow the desired path up to node gn, irrespective of the path executed after gn. This makes the test data generation more flexible and approximates it to the goal-oriented approaches.

Information to be provided to the tool are: search control parameters and data

about the program interface (number of input variables and/or parameters and corresponding types and bound values). The names of the executable program file (instrumented version) and of the file with the desired paths must also be given.

The tool takes one path at a time in the given sequence. For each desired path a search is done in the solutions base (a file with previously executed paths and the correspondent input data sets) to check if the selected path has already been executed; in such a case, the input data set that cause its execution is the desired solution and is recovered. Otherwise, input data sets whose executed paths are similar to the desired one (according to a paths' similarity metrics) are recovered to be the initial population of the Genetic Algorithm. The genetic search is done up to the generation of an input data set causing the path's execution or the identification of the likely unfeasibility of the path. Paths executed during the search and their corresponding input data sets are saved in the solutions base. The identification of the likely unfeasibility of the path is done by a dynamic heuristics which monitors the Genetic Algorithm's search progress [5].

The test data generation tool is part of the test support environment POKE-TOOL, a set of tools supporting the data flow based Potential Uses criteria [20] to test *C* programs, in its current version. These testing criteria require the execution of paths from points where variables are assigned values (definitions) to points in the program where the definitions (values) can be used. The tool can also be used to generate test data for other data or control-flow criteria (e.g. [38, 19, 31, 24, 37, 10]) by the selection of appropriate sets of paths.

5. Genetic Algorithms

Genetic Algorithms are search algorithms based on the mechanics of natural selection and genetics [12, 33, 39]. They have been applied in different areas such as machine learning, search and optimization and have been considered an efficient and robust method for solving complex problems.

These algorithms work with populations of candidate solutions to a problem, referred to as individuals. Individuals are submitted to a selection process based on each one's merit (quantified by a fitness function) and genetic operators are applied. Using these operators, the algorithm creates the next generation from the current population of solutions, by combining them and by inducing changes. The iterative combination and selection of solutions cause a continual evolution of the population towards the problem solution.

Our specific problem is to discover programs' input values that solve the restrictions imposed by the path's predicates along the desired paths. Each reached predicate must be solved and previous predicates must remain solved, despite the changes on input values in the search process. Program input variables influence path's predicates by data flow interactions that may represent complex mathematical computations (that can be as complex and diverse as problems from the real word). The Genetic Algorithm's ability in dealing with complex and diverse prob-

lems makes them a good method for searching the input values that solve path's predicates.

6. Test Data Generation Using Genetic Algorithms

The Genetic Algorithm works with a population of input data sets for the program; each input data set is a candidate solution to the problem of executing the desired path. From each new generation, in an iterative process, the input values combinations close to executing the desired path are selected. Such process leads to a progressive increase of the solutions' quality, directing the search to the program's input sub-domain associated to data sets that execute the desired path.

6.1. *Test data representation*

Each individual represents an input data set: a concatenation of values assigned to the program's input variables. All input variables are encoded as binary codes considering their types, bound values and precision. Each individual has the form of a binary string; an associated offset defines which part of the string represents which input variable.

Each type of variable is translated into a binary representation according to a codification model: floating-point variables are represented as integers considering the associated precision (by multiplying values by an adequate factor); character variables are mapped to the corresponding ASCII integer values. Integer representations of the different types (including the integer variables itself) are converted into binary codes and concatenated to form an individual. Each array element is treated and encoded as a different variable.

Each codification model has a specific decoding process for reconstituting the original values of the program input variables. Each decoded input data set is used as input for the instrumented program's execution. Dynamic information on the control and data flows during execution are used to evaluate the input data sets (compute their fitness values — the fitness function is described in Sec. 6.3).

Complex data structures require *type compatibilization drivers* (coded by the tester), software testing harnesses that are basically "type translators procedures". They receive values of variable types treated by the Genetic Algorithm (variable values that come from the decoding process) and translate them into the complex structures accepted by the program. Example 1: A driver may have 9 integer variables as input, assign their values to a 3×3 matrix of integers and make a call to the program under test with this matrix as input. Example 2: A driver may have as input one integer variable *sensor-number* and two real variables: *speed* and *temperature*, assign their values to a record structure (each variable as a field of the record), and make a call to the program under test with a pointer to this record as input. This approach allows the testing of programs with data structures such as matrices, records and linked lists as input.

6.2. *Selection, crossover and mutation*

The selection process models the nature's mechanism of survival of the fittest individuals; it is based on the fitness value associated to each input data set in the population. An input data set id_i is allocated $\frac{f_i}{favg}$ children, where f_i is its fitness value and $favg$ is the average fitness value of the population. That is, a fitter input data set (closer to executing the desired path) generates a higher number of copies in a pool of solutions to be submitted to crossover and mutation. Actually, $\frac{f_i}{favg}$ is the expected number of copies in the pool (this is the "proportionate selection scheme" [33]). This "selective pressure" provides the driving mechanism for the continual improvement of the input data sets. Moreover, the fittest input data set are preserved on the next population (with no changes) avoiding the accidental loss of good solutions ("elitist model" [12]).

Crossover (single point crossover) and mutation (simple mutation) are the genetic operators applied on the pool of solutions (pool of input data sets). Pairs of randomly chosen individuals exchange portions of their strings with a probability Pc (the crossover rate, typically 0.80), starting from a randomly selected bit. Bits are changed from 0 to 1 or vice versa with a probability Pm (the mutation rate, typically 0.03). We adopt a population size of around 80.

Crossover recombines individuals' features, mutation introduces new individuals and selection favors the survival of the fittest individuals (the ones closest to executing the desired path).

6.3. *Fitness function*

We use the following fitness function Ft to evaluate each input data set:

$$Ft = NC - \left(\frac{EP}{MEP}\right)$$

where NC, the *Program paths' similarity metrics*, is the number of coincident nodes between the executed path and the desired one, from the input node up to the node where the executed path becomes different from the desired one; it can vary from 1 to the number of nodes in the desired path. In the first case (similarity = 1), only the entry node is common to both paths; in the second case (similarity = number of nodes of the desired path), the executed and the desired paths are identical; EP is the absolute value of the predicate function [18] associated to the branch where there is the deviation from the desired path; it represents the error that causes the executed path to deviate from the desired one (for the branch $if\ (y == 100.0)$, $EP = |y - 100.0|$); MEP is the predicate function's maximum value among the input data sets that executed the same nodes of the desired path in the current population.

The fitness function reflects the fact that the greater the number of correctly executed nodes the closer is the input data set to the desired solution. Considering several input data sets executing the same number of correct nodes, the better ones

are those that present smaller absolute values for the predicate function associated to the branch where there is the deviation from the desired path. The value $\frac{EP}{MEP}$ is a penalty measure due to the input data's set error with respect to all the input data sets that executed the correct path up to the same deviation predicate in the current population. If changes in input values, when solving a given predicate, make any previous predicate along the path to be evaluated differently from the desired, they cause a lower value for NC. That is, the fitness function implicitly penalizes constraint violations of previous predicates.

The larger the fitness value the better is the input data set: fitness values are used to drive the search both to reach the desired path's nodes (increase the path's similarity values) and to solve the desired paths' predicates (decrease predicates' functions values). This fine-grained distinction among the solutions is determinant of the technique's high success rate in test data generation.

Ft is computed in three steps:

(i) values NC and EP are computed for each input data set;
(ii) MEP is computed for each predicate along the desired path;
(iii) Ft is computed for each input data set.

6.3.1. *Predicate function computation*

The predicate function values (EP) are computed using real values involved in predicates, obtained by the analysis of dynamic data flow information generated from program executions. Table 1 summarizes EP's calculation; the *Predicate* row shows possible predicate types considering the various relational operators; *Predicate Function* is the corresponding EP function and *rel* is the appropriate operator for EP *rel* 0. $LG(E1)$ refers to "positive boolean predicates" and $LG(!E1)$ to "negative boolean predicates". $k2$ is a fixed penalty for violation of boolean predicates (currently 100), $k1$ avoids division by zero when $E1 = E2$.

Table 1. Predicate functions versus relational operator.

Predicate	Predicate function	rel
$E1 > E2$	$EP = E2 - E1$	$<$
$E1 \geq E2$	$EP = E2 - E1$	$\leq$
$E1 < E2$	$EP = E1 - E2$	$<$
$E1 \leq E2$	$EP = E1 - E2$	$\leq$
$E1 = E2$	$EP = \lvert E1 - E2 \rvert$	$=$
$E1 \neq E2$	$EP = \frac{1}{(\lvert E1 - E2 \rvert + k1)}$	$\neq \frac{1}{k1}$
$LG(E1)$	$EP = k2$ if $E1 = 0$ $EP = 0$ if $E1 \neq 0$	
$LG(!E1)$	$EP = 0$ if $E1 = 0$ $EP = k2$ if $E1 \neq 0$	

When the predicate involves comparisons between characters the EP calculation is done considering ASCII values associated to the characters. In the case of string comparisons (*!strcmp(string1,string2)* in C) EP receives the sum of absolute values of the differences between the ASCII values associated to the characters in each position of the string.

In the case of compound predicates which involve logical operators, the resultant EP functions consider the sum of the functions for compounded conditions with the AND operator and the lowest value of them for compounded conditions with the OR operator [11].

The computed EP values show the real influence of the input variables in the path's predicates. No matter how complex the mathematical expressions that affect (directly or not) the predicates, it is possible to guide the search without the need for symbolic computations [6], domain-symbolic handling to compute resultant domains for arithmetic expressions from variables domains [26], or computations of predicates' slices [14].

6.4. *The test data generation process*

Figure 1 illustrates one cycle of the Genetic Algorithm in the search for test data to execute the path *1 3 4 6 7* (boldface arrows and nodes in the *CFG*). The population individuals (each binary string represents one input data set) are *decoded* to recover the input variables into their original types (integers between -7 and $+7$). Input data sets go through *type compatibilization drivers*[a] and are used in the instrumented program execution. Each execution generates dynamic information relating to control and data flows. Using this information the fitness of each input data set is computed in the *Evaluation* step.

Both input data sets $I1$ and $I2$ execute only one node along the desired path ($nc = 1$). A lower *ep* value indicates that $I2$ is closer to satisfying the predicate ($x >= 1$) than $I1$, thus, $I2$ receives a higher fitness value than $I1$. Both $I3$ and $I4$ execute the desired path up to node 4 ($nc = 3$); as $I4$ is closer to satisfying the predicate ($y == t$) it receives a higher fitness value than $I3$.

Based on the computed *fitness* values the input data sets are selected to form a pool of solutions. The *proportional selection scheme* is biased towards the better solutions (better solutions have a higher chance of being selected), favoring input $I4$ (*fitness* = 2.833, 2 copies selected), followed by $I3$ (*fitness* = 2.000, 1 copy selected). $I2$ was "lucky", despite its low *fitness* (0.250) it has 1 copy selected[b] and $I1$ (the worst solution) generates no copies. The *Crossover* operation is applied to the pool of selected solutions. It exchanges parts of the strings of pairs of randomly chosen individuals, combining input data sets $I2$ with $I4$ and $I3$ with $I4$ (generating,

[a]For this particular program the driver is not necessary since the program under test receives as input simple types, directly dealt by the genetic algorithm.

[b]Although biased toward the best input data sets, bad input data still have a chance of being selected; as in nature, they may have some good genes to contribute to the species evolution.

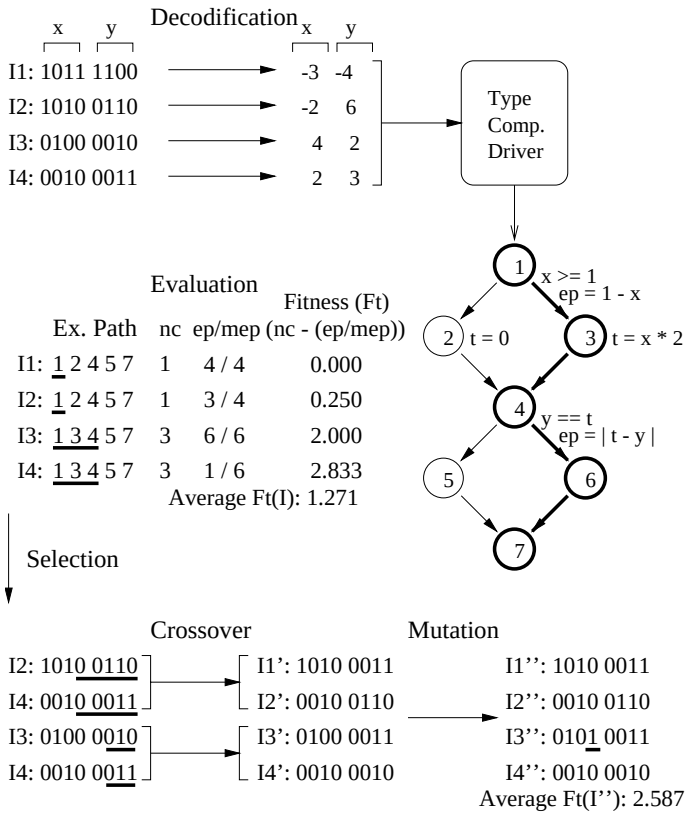

Fig. 1. Genetic algorithm's cycle.

respectively, $I1'$, $I2'$ and $I3'$, $I4'$). The *Mutation* operation is applied changing the fourth bit in $I3'$ from 0 to 1 and generating $I3''$. This new generated population ($I1'' \ldots I4''$) is the input for the next cycle of the Genetic Algorithm.

The search progress is indicated by the overall increase in the quality of solutions from the first population ($I1 \ldots I4$) to the second ($I1'' \ldots I4''$, resulted from selection, crossover and mutation of $I1 \ldots I4$), an improvement of the average fitness from 1.271 to 2.587.[c]

The cycle is repeated until a test data set that executes the desired path is found, if the path is feasible. Otherwise, if the desired path is infeasible (no test data set exists to execute it), a persistent lack of progress happens during the search as an infeasible predicate is reached. The lack of progress is identified by a dynamic heuristics which monitors the execution of the Genetic Algorithm. In this case, the tester is informed about the (likely) path unfeasibility and the predicate suspect of

[c]Value obtained by decoding and evaluation of the second population, steps not shown in the figure.

causing it [5].

In a real test data generation (*population size* = 30, average of 10 runs) solutions were found in 2 cycles, 59 program executions, 8 seconds; suggested solutions for (x, y) are: $(1, 2)$, $(2, 4)$, $(3, 6)$. Note that the program accepts 225 possible inputs (possible (x, y) combinations), but only three of them execute the desired path.

7. Instrumentation Model

The instrumented program contains file-writing statements inserted to get dynamic information on the control and data flows during the program's execution. The statements are inserted just after decision statements to save the real values involved in each predicate (expressions or variables values, relational and logical operators) and the program node executed. This information is used to compute the *path similarity* (NC) and the *predicate function* (EP).

Statements such as *error_write(N,E1,E2,OR,OL)* saves: N, node where the statement is inserted; $E1$, the first expression or variable in the predicate (left side); $E2$, the second expression or variable in the predicate (right side); OR, the relational operator involved; and OL, the logical operator connecting the next condition for composed predicates (otherwise, the flag "#" is used). For each condition in composed predicates a separate *error_write* statement is inserted.

The statement inserted in a node after a predicate allows the calculation of the predicate function associated to the node opposite to the executed one. For example, a selection command is instrumented as: $/*1*/\ if\ (i > j)\ \{/*2*/$ $error\_write(2, i, j,$ "<=","#") ...} $else\ \{/*3*/\ error\_write(3, i, j,$ ">","#") ...}, where $/*i*/$ indicates node numbers and # indicates it is a simple predicate. When node 2 is executed, the instrumentation saves information for computing the EP value relative to node 3 (or relative to branch 1–3, more precisely) and vice versa.

Case structures are treated through an instrumentation "dependent on the desired path": all nodes that are not the desired outcome in the case receive instrumentation to compute the EP value concerning the desired outcome node. It means that, irrespective of the wrong case node executed, the EP value is computed to force the execution of the desired node. This is a new treatment to case instrumentation; other approaches require cases to be represented as nested ifs.

Infinite loops are avoided by making the program execution halt if the number of traversed nodes is greater than a specified limit.

As the instrumented program is functionally equivalent to the original one, this model avoids the need for controlled executions and of inserting by hand additional constraints to deal with run-time errors in slices or modified programs' executions [11, 14] (errors can occur since points in a program can be reached which would not normally be reached). In addition, the generated output values can be directly checked against the specifications, without the need to execute the original program with the input data generated by the tool.

Currently, instrumentation is inserted by hand; an instrumentation module is being developed.

8. Reuse of Past Input Data based on Paths' Similarity Metrics

Gallagher and Narasimhan [11] performed experiments with one specially built program to assess the effect of the choice of the initial optimization search point on the performance of test data generation. Results indicate that as the distance from the initial search point to the optimal solution (that executes the path) increases, the cost of test generation also increases (with approximately linear slope). If the distance is larger than a threshold, the system fails to generate test data. They suggest repeatedly selecting random initial values in the case of failure. Other authors also select randomly the initial search point [18, 8, 14].

We improve the performance of test data generation by using knowledge on the correspondence between points on the program's input domain (input data sets) and program paths to establish an initial search region.

As we described in Sec. 4, when generating test data for a set of paths, one path is considered at a time. For each desired path a genetic search is done looking for an input data set that executes it. During the genetic search, information from the evaluation of the test data sets are kept in the solutions base. More specifically, pairs *input data set*, *program path*, one pair related to each test data set, are saved.

For a new desired path, a search in the solutions base is done and input data sets associated to paths that are similar to the desired one are recovered to compose the Genetic Algorithm's initial population. Similar paths tend to be executed by similar input data sets. Thus, input data sets of paths similar to the desired path are closer to executing it than randomly generated input data sets.

The program paths' similarity metrics (Sec. 6.3) counts coincident nodes from the entry node up to the first different node between the compared paths. A higher similarity means a greater number of predicates in common between the paths (note that the "real predicate" in terms of input variables depends on the previous data flow interactions, and so, on the previously executed nodes). Thus, the metrics estimates the distance between the input data sets in the solutions base and the desired input (that executes the desired path), by computing the similarity level between the paths on the solutions base and the desired path.

For a desired path with path length L, the algorithm searches in the solutions base paths with similarity L, $L-1$, $L-2$, ..., $Smin$ ($Smin$, minimal similarity parameter informed by the tester). If a path with similarity L is found, the associated input data set is the problem's solution; otherwise, input data sets related to paths with similarities $L-1, \ldots, Smin$, are recovered in this order. It is done until a defined fraction of the Genetic Algorithm's population size is formed. To keep a good diversity in the initial population, only a fraction of the initial population is recovered and the remaining individuals are randomly generated (in the current implementation the fraction is 0.5).

Initially the solutions base is empty and the entire initial population is randomly generated for the first desired path. However, as desired paths are processed, information is increasingly added to the solutions base. As the amount of information in the solutions base grows, the probability increases of finding input data sets closer to executing the new desired paths or even input data sets that actually execute them.

This simple technique mimics an intuitive practice of some testers: "to generate a new test to visit a code region, find a previous test that visited any neighbor code and get hints for the new test". It also fits itself to the Case-based Reasoning paradigm (CBR) [1], a problem resolution paradigm by which new problems are solved by adapting solutions used to solve similar problems: a test data set for a new desired path is obtained by identifying previously executed paths similar to it, recovering the respective test data sets, and genetically adapting them to execute the desired path.

9. Experiment and Results

An initial experiment was carried out to evaluate the tool's performance in generating test data sets and the influence of reusing past input data.

9.1. *Programs and paths*

Six small programs were used with different characteristics concerning: variable types (strings, characters, real, integers and arrays); predicates (simple, composed, involving logical variables, arrays and strings); and control structures (repetition and selection).

Altogether 40 feasible paths (14 including loop iterations; minimum, maximum and average lengths of respectively: 2, 119 and 20) were selected from these programs (5 paths from program floatcomp, 10 from quotient, 4 from strcomp, 5 from find, 8 from tritype, and 8 from expint). Short descriptions of the six programs are given below. Table 2 contains a summary of them.

Program *floatcomp*

Receives three floating point variables, performs simple computations and has some enchained *ifs* including composed predicates. Example of a path condition: $(z > y)$ *and* $(y > x)$ *and* $(z > x + y)$ *and* $(0 \leq (x * y) - z \leq 5)$.

Program *quotient*

Calculates the quotient and the remainder of the division of two positive integers. Contains an *if* statement inside a loop *while*. Some selected paths iterate the loops many times (path lengths up to 49 nodes) executing different sub-paths inside the loops in each iteration.

Program *strcomp*

Receives three characters and a string with five positions. Has predicates such as: $if(ch1 == `a')$ and $if(!strcmp(name, "test1"))$. One of the desired paths requires

Table 2. Programs characteristics.

Program	Input type	Number of nodes	Cyclomatic complexity
floatcomp	3 float	10	5
quotient	2 integer	13	6
strcomp	3 char + 1 string [5]	9	5
find	2 integer + array of integer	22	9
tritype	3 integer	22	9
expint	1 integer + 1 float	30	16

as input the data set ("a","b","c","test1") to be executed. For this path the test data generation fails in some attempts (on execution mode [ii], see Sec. 9.2). In these situations the Genetic Algorithm explores solutions close to the desired one but does not reach the desired solution within the upper bound of search cost. Note that each char variable may be assigned 128 possible values (possible characters in ASCII Table), giving an input domain of 128^8 ($7.21 * 10^{16}$) input data sets, among which only one executes the path. This illustrates that a very simple program can be designed to present a very hard test data generation problem.

Program *find*

Accepts an input array A of integers with N elements and an index F; returns the array with every element to the left of $A[F]$ less than or equal to $A[F]$ and every element to the right of $A[F]$ greater than or equal to $A[F]$. It has enchained loops, logical predicates (e.g., $if(!b)$) and some with comparisons between array elements (e.g., $(while(a[i] < a[f]))$). Some of the selected paths are long (up to 119 nodes) and are executed only for very special input characteristics (e.g., all elements of A with the same value).

Find has a "dynamic interface" (the number of program input variables is itself defined by an input variable — N). To deal with this characteristic the Genetic Algorithm works with the defined upper bound of the number of positions for A, but calls the program under test passing only the N variables (N defined in execution time), i.e., not all encoded variables really influence the execution. This solution allows the test data generation, but introduces some "noise" that degrades the Genetic Algorithm's performance. Other authors test this program using fixed values for N, a simpler solution but that may induce unfeasibility of the (otherwise feasible) selected paths in the program.

If $F > N$ the program enters an infinite loop. This bug does not disturb the test data generation because program instrumentation is such that the program halts in these situations.

Program *tritype*

Accepts three integers representing lengths of a triangle's sides (a, b, c). Classifies the type of triangle and calculates its area. For executing the path that classifies a rectangle triangle an input data set must satisfy: $(a \geq b)$ and $(b \geq c)$ and $(a < (b + c))$ and $(a \neq b)$ and $(b \neq c)$ and $(a^2 = b^2 + c^2)$ [d].

Program *expint*

Accepts an integer (n) and a float variable (x). It has enchained *ifs*, *for* loops, complex computations (e.g., $ans = (nm1! = 0?1.0/nm1 : -log(x) - EULER)$, ans = exp(-x)/x) and predicates (e.g., $if((n < 0)||(x < 0.0)||(x == 0.0\&\&(n == 0.0||n == 1)))$, $if(x == 0.0)$).

Some of the selected path are executed only with very specific inputs. Example: a path is executed for $n = 2 * 10^9$ and $x = 1 * 10^{-9}$, but is not executed for the same n but $x \geq 1 * 10^{-8}$ or $x = 0$.[e]

For this program, some attempts to generate test data fails if the precision of variable x or if the upper bound of n are below the necessary to execute the selected paths. In these situations the values explored during the search converge to the established limits (of precision and magnitude) allowing the tester to see the necessity of establishing new upper bound and precision.

9.2. *Cost and effectiveness in automatic test data generation*

Three execution modes are defined:

Mode [i]: use of the Genetic Algorithm to search for input data sets. Reuse of past input data based on paths' similarity metrics to compose the Genetic Algorithm's initial population.

Mode [ii]: use of the Genetic Algorithm to search for input data. Entire initial population is randomly generated.

Mode [iii]: use of a random data generation function integrated into the tool.

Average numbers of program executions needed to generate test data were computed for each path and execution mode considering 10 attempts. Upper bounds on the number of executions were defined (100,000 executions for programs find, tritype, and expint — more complex — and 45,000 executions for other programs). When the upper bounds were reached a search failure was detected and these bounds were used to compute average values.

The average number of executions was computed for each program considering all the selected paths in the program (AE values). AE reflects the average number of program executions required for generating a test data set for each selected path in the program with the respective execution mode (see Table 3).

[d]For generating test data for this path the genetic search takes (on average) 2,135.0 program executions and the random search 21,566.0 program executions.
[e]For generating test data for this path the genetic search takes (on average) 14,368.6 program executions and the random search always fails for the limit of 100,000.0 program executions.

Table 3. Cost for test data generation (AE values).

Program	Execution mode [i]	Execution mode [ii]	Execution mode [iii]
floatcomp	98.1	101.8	896.4
quotient	99.9	195.5	6909.1
strcomp	4269.4	17502.1	45000.0
find	6858.5	8153.7	41589.8
tritype	373.0	608.4	3861.4
expint	2463.4	4154.0	38820.8

Execution mode [i]: 100% of success was achieved. An average of 2,360.4 program executions was required per attempt, equivalent to approximately 7 minutes of search.[f]

Execution mode [ii]: 98.2% of success was achieved. An average of 5,119.2 program executions was required per attempt, equivalent to approximately 14 minutes of search.

Execution mode [iii]: 70.0% of success was achieved. An average of 22,849.1 program executions was required per attempt, equivalent to approximately 67 minutes of search.

9.3. *Influence of reusing past input data*

The reuse of past input data, resulted in a total average reduction of 53% in the number of program executions to find input data sets. This reduction confirms empirical results from [11] concerning the influence of the choice of initial points in the search, extends their validity to other programs (in [11] only one program was used) and to the search using Genetic Algorithms. The cost reduction resulted from: (i) input data sets that execute the desired paths found on the solutions base (22 of the 40 paths); (ii) reduction of the genetics' search cost (for 18 paths, input data sets that execute similar paths were recovered, reducing the search cost in 28%). The cost of searching the solutions' base was always negligible compared to the cost of the genetic search.

The success rate was 100% when reusing past input data against 98.2% when the initial population was entirely randomly generated. The chance of success before reaching the upper bounds are higher since the average search cost is reduced.

10. Conclusions and Future Work

This work focus on the automation of test data generation, a critical task in structural testing and a "bottleneck" for the full automation of structural testing.

[f]In a Sun Ultra-1 workstation; 256Mb RAM; clock of 143 MHz; operating system SunOS 5.5.1. The implementation do not emphasize performance (e.g., communication between the tool and the instrumented program uses sequential files).

The major contributions are:

(i) a new technique for test data generation for path testing using genetic algorithms;
(ii) the combination of data and control flow information in a fitness function that guides the search more accurately than previous approaches;
(iii) use of a technique for defining initial search region to improve the performance of the test data generation.

Ongoing and future work include: incorporating advances from Genetic Algorithms applications on constraint optimization problems [39]; investigation of specialized genetic operators for test data generation (e.g., for efficient solution of linear predicates as in [14]); adaptation of the technique for goal oriented and error-based test data generation and for integration testing; and, conducing empirical evaluations with more complex programs.

Acknowledgments

The authors would like to thank the contributors Ivan Ricarte, José Carlos Maldonado and members of the DCA/FEEC Test Group as well as the anonymous referees for their suggestions.

References

1. A. Aamodt and E. Plaza, "Case-based reasoning: Foundational issues, methodological variations, and system approaches", *AICom — Artificial Intelligence Communications* **7(1)** (1994) 39–59.
2. A. Bertolino and M. Marré, "Automatic generation of path covers based on the control flow analysis of computer programs", *IEEE Trans. on Software Engineering* **20(12)** (1994) 885–889.
3. D. L. Bird and C. U. Munoz, "Automatic generation of random self-cheking test cases", *IBM System Journal* **22(3)** (1983) 229–245.
4. R. S. Boyer, B. Elspas, and N. L. Karl, "Select: A formal system for testing and debugging programs by symbolic execution", *ACM SigPlan Notices* **10(6)** (1975) 234–245.
5. P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data", in *Proc. IEEE Int. Conf. on Automated Software Engineering* (*ASE*2000), IEEE, Grenoble, France, September 2000, pp. 209–218.
6. L. Clarke, "A system to generate test data and symbolically execute programs", *IEEE Trans. on Software Engineering* **2(3)** (1976) 215–222.
7. R. A. De Millo and A. J. Offutt, "Constraint-based automatic test data generation", *IEEE Trans. on Software Engineering* **17(9)** (1991) 900–910.
8. R. Ferguson and B. Korel, "The chaining approach for software test data generation", *ACM Trans. on Software Engineering and Methodology* **5(1)** (1996) 63–86.
9. I. Forgács and A. Bertolino, "Feasible test path selection by principal slicing", *Software Engineering Notes* **22(6)** (1997) 378–394.
10. F. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria", *IEEE Trans. on Software Engineering* **14(10)** (1988) 1483–1498.

11. M. J. Gallagher and V. L. Narasimhan, "Adtest: A test data generation suite for *Ada* software systems", *IEEE Trans. on Software Engineering* **23(8)** (1997) 473–484.
12. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
13. N. Gupta, A. P. Mathur, and M. L. Soffa, "Automated test data generation using an iterative relaxation method", *Foundations of Software Engineering* **11** (1998) 231–244.
14. N. Gupta, A. P. Mathur, and M. L. Soffa, "*UNA* based iterative test data generation and its evaluation", *Proc. IEEE Automated Software Engineering Conference*, October 1999.
15. W. E. Howden, "Symbolic testing and dissect symbolic evaluation system", *IEEE Trans. on Software Engineering* **3(4)** (1997) 266–278.
16. D. C. Ince, "The automatic generation of test data", *The Computer Journal* **30**(3) (1987) 63–69.
17. B. F. Jones, H. H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms", *Software Engineering Journal*, September 1996, pp. 299–306.
18. B. Korel, "Automated software test data generation", *IEEE Trans. on Software Engineering* **16(8)** (1990) 870–879.
19. J. W. Laski and B. Korel, "A data flow oriented program testing strategy", *IEEE Trans. on Software Engineering* **9(3)** (1983) 347–354.
20. J. C. Maldonado, M. L. Chaim, and M. Jino, "Bridging the gap in the presence of infeasible paths: Potential uses testing criteria", in *Proc. XII Int. Conf. of the Chilean Computer Science Society*, Santiago, Chile, October 1992.
21. C. C. Michael and G. McGraw, "Automated software test data generation fo complex programs", in *Proc. Int. Conf. on Automated Software Engineering*, IEEE, Honolulu, Hawai, October 1998.
22. C. C. Michael, G. E. McGraw, M. A. Schatz, and C. C. Walton, "Genetic algorithms for dynamic test data generation", in *Proc. Int. Conf. on Automated Software Engineering* (*ASE '97*), IEEE, November 1997.
23. W. Miller and D. L. Spooner, "Automatic generation of floating-point test data", *IEEE Trans. on Software Engineering* **2(3)** (1976) 223–226.
24. S. C. Ntafos, "On required element testing", *IEEE Trans. on Software Engineering* **10(6)** (1984) 795–803.
25. S. C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing", *IEEE Trans. on Software Engineering* **5(5)** (1979) 520–529.
26. A. J. Offut, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation", *Software Practice and Experience* **29(2)** (1999) 167–193.
27. M. A. Ould, "Testing — a challenge to method and tool developers", *Software Engineering Journal*, March 1991, pp. 59–64.
28. R. P. Pargas, Harrold M. J., and R. R. Peck, "Test-data generation using genetic algorithms", *Journal of Software Testing, Verification and Reliability* **9** (1999) 263–282.
29. L. M. Peres, S. R. Vergilio, M. Jino, and J. C. Maldonado, "Path selection strategies in the context of software testing criteria", in *1st Latin American Test Workshop*, IEEE, Rio de Janeiro, Brazil, January 2000, pp. 222–227.
30. C. V. Ramamoorthy, F. H. Siu-Bun, and W. T. Chen, "On automated generation of program test data", *IEEE Transactions on Software Engineering* **2(4)** (1976) 293–300.
31. S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information", *IEEE Transactions on Software Engineering* **11(4)** (1985) 367–375.
32. M. Roper, I. Maclean, A. Brooks, J. Miller, and M. Wood, "Genetic algorithms and

the automatic generation of test data", *Technical Report RR/95/195 Dep. Computer Science, University of Strathclyde*, 1995.
33. M. Srinivas and L. M. Patnaik "Genetic algorithms: A survey", *IEEE Computer* **27(6)** (1994) 17–26.
34. N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing", in *Proc. Int. Symp. on Software Testing and Analysis*, ACM, Florida, USA, March 1998, pp. 73–81.
35. N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation", in *IEEE Automated Software Engineering*, IEEE, Honolulu, 1998.
36. N. Tracey, J. Clark, and K. McDermid Mander, "Automated test-data generation for exception conditions", in *Software-Practice and Experience* **30** (2000) 61–79.
37. H. Ural and B. Yang, "A structural test selection criterion", *Information Processing Letters* **28(3)** (1988) 157–163.
38. W. E. Herman, "Flow analysis approach to program testing", *The Australian Computer Journal* **8(3)** (1976) 259–266.
39. M. Zbigniew, *Genetic Algorithms + Data Strutures = Evolution Programs*. Springer Verlag, 3rd edn., 1996.