Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data

Paulo Marcos Siqueira Bueno
DCA/FEEC/Unicamp
Campinas - S.P. - Brazil
e-mail: bueno@dca.fee.unicamp.br

Mario Jino
DCA/FEEC/Unicamp
Campinas - S.P. - Brazil
e-mail: jino@dca.fee.unicamp.br

Abstract

A tool and techniques are presented for test data generation and identification of a path's likely unfeasibility in structural software testing. The tool is based on the Dynamic Technique and search using Genetic Algorithms. Our work introduces a new fitness function that combines control and data flow dynamic information to improve the process of search for test data. The unfeasibility issue is addressed by monitoring the Genetic Algorithm's search progress. An experiment shows the validity of the developed solutions and the benefit of using the tool.

Keywords: test data generation, dynamic technique, path unfeasibility, genetic algorithms.

1 Introduction

Structural testing criteria are used to select program structural components (named required elements) to be exercised by the tester. To satisfy a criterion means to exercise all the structural components required by the criterion through execution of the program. Selecting test data to exercise structural components is an extremely complex task as it requires a careful analysis of the program code, mental effort and knowledge of the underlying concepts of the testing criterion. Hence, automation of test data generation may provide a significant cost reduction in software testing.

The test data generation process may consist of two steps: [i] selection of complete paths in the program to cover a given required element; and [ii] generation of test data to execute complete paths. Path selection for the satisfaction of structural testing criteria (step [i]) has been studied by many authors [18, 5, 1, 21]. The symbolic execution [3, 14] and the dynamic technique [16, 4, 8, 11] are approaches for the generation of test data to execute program paths (step [ii]).

In the symbolic execution approach the execution conditions for a given path are modeled as a function of the program's input variables. Algorithms that search for solutions satisfying the conditions and causing the execution of the intended path make use of this symbolic representation. This approach imposes restrictions to the treatment of loops, composed variables and function calls.

The dynamic technique is based on the program's real execution, numerical methods and dynamic data flow analysis. Actual values are assigned to input variables and the program's execution flow is monitored. If an undesired branch is taken (that is, the execution flow deviates from the intended one), function optimization or iterative relaxation methods are used to determine values for the input variables which make the right branch to be taken.

A crucial question in structural software testing is the existence of infeasible paths, for which there are no input data that make them be executed. Determination of these paths is an undecidable question. Commonly used methods for this determination are partial, providing solutions only for particular cases.

Previous works on unfeasibility determination are based on symbolic processing and static data flow analysis. Other authors use control, data flow or branch correlation analysis for path selection. Numerical analysis has been recently proposed to identify infeasible paths.

We present a tool and techniques for the automation of test data generation and infeasible path identification. Dynamic technique [16] and search based on Genetic Algorithms [10] are used for test data generation. A dynamic heuristics is proposed for identification of likely infeasible paths. Our approach is applicable regardless of requirements on tractable path equations,

0-7695-0710-7/00 $10.00 © 2000 IEEE

limitation inherent to the previous approaches. A case study is presented detailing the heuristics proposed for unfeasibility detection.

The following are the major contributions of this work: 1) use of a fitness function that combines control and data flow information in the Genetic Algorithm to generate test data to execute program paths, and 2) identification of likely unfeasibility of a given path through a dynamic heuristics which monitors the execution of the Genetic Algorithm.

2 Basic concepts

The structure of a program P is represented by a directed graph called *Control Flow Graph (CFG)* $G = (N, E, s, e)$, where N is a set of nodes, E is a set of edges, s is the unique entry node, and e is 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 [17]. An edge (n_i, n_j) is called a branch if the last statement of n_i is a selection statement or a repetition statement. A branch predicate can be assigned to a branch.

A sub-path is a node sequence $(n_1, n_2, ..., n_k)$, $k \geq 2$, such that an edge exists from n_i to n_{i+1}, $1 \leq i \leq (k-1)$ (that is, $(n_i, n_{i+1}) \in E$).

A path or complete path is a sub-path where the first node is the entry node s and the last node is the exit node e. Moreover, an intended path is a path we want to execute.

An input variable x_i of a program P is a variable which appears in an input statement, an input parameter, or is a global variable used in the program. The type of an input variable can be any of all the different types of the programming language.

$IA = (x_1, x_2, ..., x_n)$ is an array of input variables of the program P.

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 $D = D_{x1} \times D_{x2} \times ... \times D_{xn}$, where D_{xi} is the input domain of the input variable x_i.

A single point x in the n-dimensional input space $x \in D$ is referred to as an input or test input.

A path is feasible (or executable) if there exists an input $x \in D$ for which the path is traversed during program execution; otherwise, the path is infeasible (or unexecutable).

Given that $CP = (n_1, n_2, ..., n_k)$ is an intended path, the objective is to find $x \in D$ which makes CP to be executed or to determine the likely unfeasibility of CP, if this is the case.

3 Related work

Hedley and Hennell [13] discuss the main causes of infeasible paths in programs and present a classification for the causes.

Clarke [3] presents a system which generates test data and symbolically executes program paths to aid in the selection of test data. Linear programming techniques are used to generate input data for paths whose constraints are linear. Symbolic execution is used to generate the constraints representing the selected path and an inequality solver is used to solve the linear constraints; if the solver finds a constraint inconsistent with the previous ones the path is shown to be nonexecutable.

Frankl and Weyuker [6, 7] discuss concepts related to unexecutable paths and their consequences on the application of data flow testing criteria; they present a heuristics using data flow analysis and symbolic execution techniques to determine infeasible data flow associations.

Malevris et al [18] use the number of predicates in a path to predict unfeasibility. They conclude that the greater the number of predicates in a path, the greater the probability of it being infeasible.

Goldberg et al [9] use symbolic evaluation to construct a formula that is solvable if and only if there are input values which drive the execution down the path. A theorem prover is invoked to test the feasibility of the formula.

Korel [16] 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 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 in the test data generation process (the "chaining approach") [4]. If an undesirable execution flow is observed in any branch and the search process cannot find an input value to change this execution flow, data flow analysis is applied to identify nodes that must be executed prior to reaching the branch to increase the chance of altering the flow execution.

Vergilio et al [26] performed experiments using C programs and present a classification of infeasible paths within the context of data flow testing criteria. They also explore the effect of context dependency on unfeasibility of paths of a software unit.

Forgács and Bertolino [5] use control flow information and data dependencies to select feasible paths reaching a given point in the program. The approach minimizes the number of predicates which are essential

to reach the point, reducing the number of predicates examined during input data generation.

Bodík et al [1] use static branch correlation analysis to define a technique for identification of infeasible program subpaths and infeasible def-use pairs.

Tracey et al [25] present a test-case data generation framework based on optimization technique called Simulated Annealing and illustrates the aplication of this framework to test specification failures and exception conditions.

Gupta et al [11] suggest a method where, if the intended path is not executed, the input is iteratively refined. Two representations are computed for each predicate: the "slice", a set of commands that influence the predicate along the intended path; and the predicate's linear arithmetic representation as a function of input variables. The two representations are used to obtain a desired input by iteratively refining the initial input value. This technique is improved [12] through a "Unified Numerical Approach" where the choice of input values is done using Least Square Errors techniques. Numerical analysis is applied to identify infeasible paths when all the predicates in the paths are linear with respect to the input variables.

The following proposals make use of the Dynamic Technique and of optimization using Genetic Algorithms, search algorithms based on genetic mechanisms [10]. Search is directed by a fitness function that is used to select the best solutions.

In the Jones et al's approach [15] a low fitness value is given to the candidate solution if the branch to be executed is not reached. If the branch is reached but not executed, the fitness value measures the difference between the predicate value needed to execute the branch and that of the candidate solution. Michael et al [19] have a similar approach; however, the attempt to execute a given branch is delayed until the input data reaching the branch are found. Notice that in both works the fitness functions do not distinguish which candidate solutions are "closer" to reaching the branch. This makes the search poorly directed, given that input data executing paths "far" or "close" to reaching the branch get the same fitness value.

Pargas et al apply a Genetic Algorithm to search for test data that exercise program "targets" (statements or branches in the current version) [20]. A fitness function is used to qualify each solution according to the number of executed predicates with regard to the *Control-Dependence Graph (CDG)* predicates. This graph represents, for each node in the CFG, an acyclic path that contains the predicates that must be satisfied to cause the node's execution. The approach assumes that test data are "closer" to executing a target when they satisfy a higher number of predicates in the CDG path of the node. However, the fitness values do not take into account which candidate solution is "closer" to satisfying the predicates. For example, a candidate solution reaching a predicate "$if(x = 0)$" with $x = 2$ is clearly better than one reaching it with $x = 10$ (in the sense that the first solution is closer to satisfying "$if(x = 0)$" than the second one). This fact also makes the search poorly directed, since all the candidate solutions reaching a given predicate get the same fitness value; it makes no difference how "far" or "close" to solving the predicate they are.

Roper et al [23] present an aproach where the population of the Genetic Algorithm is the set of test data (each individual is a test data of the set) and the fitness of an individual corresponds to the coverage achieved in the program under test. The algorithm evolves the population to achieve a desired level of branch coverage.

4 The proposed technique

The tool described here aims at generating input data for the execution of complete paths in the program under test and to identify path unfeasibility when this is the case. The tool uses the dynamic technique [16] for input data evaluation and a Genetic Algorithm [10, 24, 27] for input data selection, aiming to find the data that make the intended path to be executed.

Initially, the program is instrumented by a selective insertion of statements which provide information on control and data flow during the program's execution.

Information to be provided are the search control parameters and data about the program's interface (number of input variables and/or parameters and corresponding types and bound values). The names of the executable program file (instrumented and compiled version) and of the file with the intended paths must also be given.

Our approach combines previous works using Genetic Algorithms ([15, 19] and [20]), and introduces a new fitness function using control and data flow information to drive the search with a higher precision. Our fitness function evaluates solutions considering the number of correctly executed nodes with regard to the intended path (control flow information as in [20]) and the values associated to the expressions involved in predicates (dynamic data flow information as in [15] and [19]). This is done by using a path similarity metrics and Korel's predicate function [16]. The fitness function provides the Genetic Algorithm precise information to be used to drive the search both to reach the path's nodes and to solve the path predicates.

The precision of our fitness function makes it possible to verify the strong correlation between the lack of search progress and the unfeasibility of the path, which is the basis of the approach proposed for unfeasibility detection.

The test data generation tool is part of the testing suport environment POKE-TOOL [2], a set of tool supporting the data flow based Potential Uses criteria [17] to test C programs, in its current version. These testing criteria require that the tester executes paths from points where variables are assigned values (definitions) to points in the program where the definitions (values) can be used. The paths are automatically selected according to the strategies defined in [21], aiming to satisfy these criteria.

4.1 Search for test data based on genetic algorithms

Genetic Algorithms are search algorithms based on evolution, natural selection and genetic mechanisms [10, 24, 27]. Genetic Algorithms have been applied in different areas related to machine learning and function optimization and have been considered efficient and robust search and optimization methods.

These algorithms work with populations of potential solutions to a problem, referred as individuals. These solutions are submitted to a selection process (the reproduction) based on each individual's merit (quantified by the 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 population's evolution towards the problem solution.

Each individual represents a program's input data, that is, a possible problem solution. The reproduction is based on this population and it means to select the best solutions (that is, input data closest to executing the intended path). The "proportionate selection scheme" [24] is used to make a fitter solution yield a higher number of offspring. Essential genetic operators (simple mutation and single point crossover) are applied to the selected solutions with probabilities Pm (typically 0.03) and Pc (typically 0.80), respectively. Population size is around 80. The "elitist model" [10] is also adopted to stress preservation of the best solutions in the next generation.

The Genetic Algorithm's initial population is generated using input data from a "solutions base" which contains complete paths executed in previous processes of test data generation (for other intended paths) and the corresponding input data. The *path similarity metrics* (Section 4.1.1) is used to measure the similarity between a new intended path and the paths on the "solutions base". If the intended path is already on the base the respective input data (problem solution) is recovered. Otherwise, the input data associated to paths with higher similarity level are recovered first.

The tool directly supports primitive types of input variables, that is, integer, real and character. Monodimensional arrays of these types are also treated, which allows arrays of characters as input. Complex data structures require *type compatibilization drivers* (provided by the user), whose function is to receive the input variables decoded from types treated by the genetic algorithm and translate them into the complex structures accepted by the program; it allows the tool application to test programs that have as input structures such as arrays and records.

Input variables are encoded as binary codes. An offset is calculated and assigned to individuals considering types, bound values and precision of those variables in the program. This offset is used in the individual decodification process needed for an individual's evaluation. Real type variables are converted to integers considering the associated precision. Character type variables are mapped to the corresponding ASCII integer values. Each element in arrays is encoded in an independent form.

4.1.1 Fitness function

The evaluation of a population aims to select the best solutions, carried out for each Genetic Algorithm generation, and is done using the information from the program's execution with the input data encoded in the individual. The evaluation outcome is the association of a fitness value to each individual.

The fitness function Ft used to evaluate each candidate solution has the form:

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

where:

NC is the value of the *Path similarity metrics* computed considering the number of coincident nodes between the executed path and the intended one, from the entry node up to the node where the executed path is different from the intended one; this value can vary from 1 to the number of nodes in the intended 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 intended path) the executed and intended paths are identical;

EP is the absolute value of the predicate function associated to the branch where there is the deviation from the intended path. This value reflects the error that causes the executed path to deviate from the intended one;

MEP is the predicate function maximum value among the candidate solutions that executed the same nodes of the intended path.

The fitness function reflects the fact that the greater the number of correctly executed nodes the closer is the candidate solution to the desired solution. From the several solutions with the same number of correct nodes, the most adequate are those with smaller absolute values for the *predicate function* [16] associated to the branch where there is deviation from the intended path. This function indicates the error that causes the deviation and measures how distant is the candidate solution from executing the correct branch.

Notice that the value $\frac{EP}{MEP}$ is a measure of the candidate solution error with respect to all the solutions that executed the right path up to the same deviation predicate. This value is used as a solution penalty. Thus, the search dynamics is characterized by the coexistence of two objectives: maximize the number of nodes correctly executed with respect to the intended path and minimize the *predicate function* of the reached predicates.

This fitness function makes a very precise distinction between all the input data in the population. It puts together control and data flow dynamic information to quantify the merit of each input data considering the intended path execution. The fitness value of a given candidate solution indicates its survival chance, determining its influence on the next generations.

4.1.2 Predicate function calculation

The predicate function EP is obtained by dynamic data flow analysis. Each simple predicate $E1$ *op* $E2$ is transformed into the form EP *rel* 0, where $rel \in \{<, \leq, =, \neq\}$. For example, the predicate $a > c$ is transformed to $c - a < 0$. EP is actually a function (directly or indirectly) of program input variables; thus, changes on these variables have the potential of influencing this function's value. Thus, it is possible to manipulate input variables to minimize a given value of predicate EP. Minimizing EP means to progress in the direction to satisfy it.

Table 1 summarizes EP 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 to violation of boolean predicates (currently 100), $K1$ avoids division by zero when $E1 = E2$ (currently 0.3).

Table 1: Predicate function according to the type of relational operator involved

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 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 [8].

4.1.3 Illustration of the test data generation process

Figure 1 shows the test data generation process to execute the path 1 3 4 6 7. The various processes related to the genetic algorithm cycle are shown. The individuals are *decoded* and the original type input values are recovered (integers between -7 and +7). Input data generated from one individual decoding go through (if necessary) *type compatibilization drivers* and are used in the program execution. Information is obtained then to calculate the fitness of an individual, done in the *Evaluation* step. The genetic operators *selection, crossover and mutation* are applied to the current population generating a new population for the next iteration cycle of the genetic algorithm.

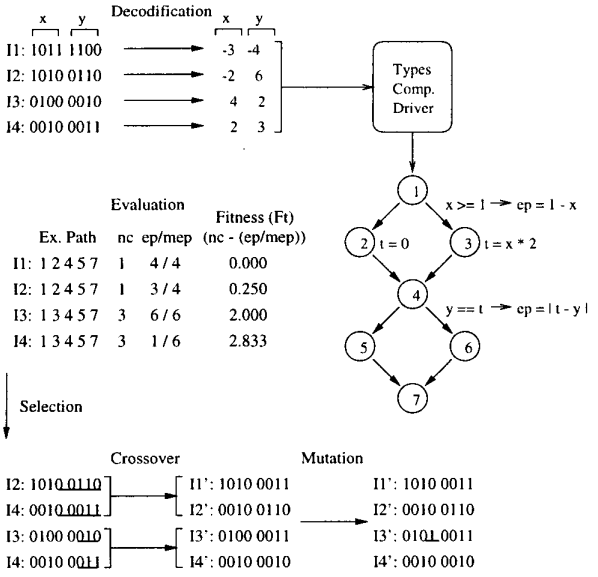

Figure 1: Genetic algorithm application in the tool

4.2 A dynamic approach to unfeasibility detection

In our approach the Genetic Algorithm's search progress is monitored to address the unfeasibility question. The Genetic Algorithm's search happens with a solutions' population that evolves according to probabilistic rules. This evolution can be evaluated by monitoring the performance measures. This approach is used in the context of genetic algorithm's function optimization [10]. In our context monitoring is useful to identify the *potential unfeasibility* of an intended path.

The *population's best fitness* reflects the quality of the best population's solutions and can be used to check in a very reliable form the search progress. When generating test data for feasible paths, a continual population's best fitness improvement can be observed as the path's predicates are reached and solved by the Genetic Algorithm. On the other hand, attempts to generate test data for infeasible paths result, invariably, in a persistent lack of progress concerning this value because there is an infeasible predicate (with respect to the path) that makes impossible the search progress. In this situation some of the individuals in the population encode input data that make the infeasible predicate to be reached, but none of them (of course) succeeds in solving the corresponding predicate function.

The search progress monitoring is a way to detect as soon as possible an infeasible path by using the dynamic technique. Another alternative solution is to finish the search at a pre-defined time-out [20]. However, in this case the chance is higher of finishing the search when trying to generate data for a "hard" feasible path or of spending too much effort in trying to generate data for an infeasible path that can be "easily" identified by using progress monitoring.

The lack of progress identification is done by the following heuristics:

If

I) $aj[i] - aj[i-1] \leq \delta \; \forall \; i, j \leq i \leq (j + NL)$ and
II) $aj[j + NL] - aj[j] \leq \Delta$

then: *Potential unfeasibility* situation identified.

Where: $aj[i]$ is the best population's fitness in generation i; $aj[i-1]$ is the best population's fitness in generation $i-1$; j is the generation in which the relation I was initially observed; δ is the lower bound used to consider a lack of progress situation in two successive generations; Δ is the lower bound used to consider a lack of progress situation accumulated through NL generations; and NL is the number of generation required to satisfy condition I.

This heuristics is implemented through the monitoring of the best population's fitness values. In each new generation this value is compared to that of the previous generation. If the increase in the best fitness is lower than δ, a counter is incremented. If the counter reaches NL, rule II is applied. In every generation, if the increase of the best fitness is greater than δ, the counter is set to 0.

Rule II is applied whenever the search presents successive progresses lower than δ through NL generations. It is then verified if since the start of the counting (generation j) up to the current generation $(j + NL)$ the search progress was lower then Δ. In this case the potential unfeasibility of the intended path is identified.

The NL value determines how persistent the lack of search progress need be to alert the tester about the likely unfeasibility of the path. In more complex data generation problems, it is desirable to associate higher values to NL as a tolerance in the lack of progress, acceptable because of the higher complexity. This is done through an equation that models the problem complexity as a function of the number of input variables and the number of nodes in the intended path. It estimates the complexity of test data generation concerning the specific problem and automatically adjusts the heuristics to make it possible to distinguish the fine difference

between the impact on the search dynamics of unsolvable path's constraints and of "hard" (solvable) path's constraints.

Gallagher and Narasimhan [8] found that the test data generation cost grows with the number of input variables and with the path length. We use this result to establish the following equation:

$$NL = Aj1 + Aj2 \times (Np + Nv)$$

Where: NL is the required number of generations to satisfy condition [I]; $Aj1$ is a constant; Np is the number of nodes in the intended path; Nv is the number of program input variables; $Aj2$ is a parameter which controls the influence of Np and Nv in NL.

The following values are being used for the parameters: $\delta = 0.30$; $\Delta = 0.20$; $Aj1 = 30$ and $Aj2 = 3$.

When the potential unfeasibility of the intended path is identified by the heuristics the tester is provided with the predicate suspect of causing path unfeasibility (along the intended path) and the values δ, Δ and NL. The suspect predicate is obtained by examining the current Genetic Algorithm population. Due to the "elitist model" the best solution in the current population is also the best within the entire search. Therefore, the last predicate this solution reaches is also the last reached in the path throughout the entire search, making it the sole suspect of causing path unfeasibility.

It is not possible to be 100 % sure about unfeasibility of a path determined this way; there will always be the question: adequate input values were not found because of search deficiency or such values do not exist (i.e., the path is infeasible) ? Nonetheless, it is important to remark that the identification done by the tester can equally yield wrong decisions. If unfeasibility is identified for a path, the tester can decide if he relies on this identification or if he tries to confirm it by analyzing the code; in this case, the predicate suspect of originating the path unfeasibility can be useful in his analysis.

5 Experiment and results

An initial experiment was carried out to evaluate the tool's performance in generating test data and the cost effectiveness of the dynamic heuristics for identification of potential unfeasibility.

Six small programs were used with different characteristics of variables types (strings, characters, real, integers and arrays); predicates (simple, composed, involving logical variables, arrays and strings) and control structures (repetition and selection), including the programs *quotient* [8], *find* [6], *tritype* [22] and *expint* [11]. Table 2 summarizes the programs' characteristics.

Table 2: Programs' characteristics

Charac. / 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

Two execution modes are defined:

Mode [i]: use of the Genetic Algorithm to search for input data and the recovery of solutions from the "solutions base" to compose the Genetic Algorithm's initial population;

Mode [ii]: use of a random data generation function.

Altogether 40 feasible paths were selected from these programs (5 paths from program floatcomp, 10 from program quotient, 4 from program strcomp, 5 from program find, 8 from program tritype, and 8 from program expint). To reduce random variations in the results, 10 attempts for each execution mode and each program path were taken to generate test data. For each attempt the number was recorded of program executions needed to obtain input data which executed the intended path (referred as NPE). If the upper bound on number of executions was reached the search was finished, the search failure was detected and this limit was assigned to NPE. This upper bound on the number of executions was the same for both execution modes (100,000 executions for programs find, tritype, and expint — more complex — and 45,000 executions for programs strcomp, floatcomp and quotient).

The average value of NPE for each path and each execution mode was computed considering the 10 attempts (referred as APE value). The total average of APE values for each program was computed considering all the selected paths in the program (referred as TAPE values). TAPE values reflect the average cost of executing each selected path in the program (in terms of required number of program executions) with the respective execution mode. Table 3 summarizes TAPE values for each program and execution mode.

100% of success was achieved by using the Genetic Algorithm and by recovering the solutions from the

Table 3: Programs and results

TAPE values / Program	Execution mode [i]	Execution mode [ii]
floatcomp	98.1	896.5
quotient	99.9	6909.1
strcomp	4269.4	45000.0
find	6858.5	41589.8
tritype	373.1	3861.5
expint	2463.4	38820.9

"solutions base" to compose the initial population. An average of 2,360.41 program executions was required in each attempt, equivalent to approximately 7 minutes of search (using a Sun Ultra-1 work station; 256Mb RAM; clock 143 MHz; operating system SunOS 5.5.1).

70% of success was achieved by using the random data generation. An average of 22,849.11 program executions was required in each attempt, equivalent to approximately 67 minutes of search.

5.1 Cost and effectiveness of the potential unfeasibility dynamic identification heuristics

12 infeasible paths were selected from these programs to evaluate the cost of the unfeasibility identification. 10 attempts were done to generate test data for each path. The average value of the number of program executions needed for unfeasibility identification was 7,706.04, equivalent to approximately 23 minutes of search. This cost is influenced by the NL values computed for each path and program and by the position of the infeasible predicate along the path (deeper predicates need more executions to be reached).

The heuristics has yielded evaluation errors in 1.25% of the cases. In these situations, feasible paths were erroneously identified as infeasible. This happened because hard to solve predicates (that establish severe restrictions in the program's input domain) cause a persistent lack of search progress, causing the incorrect unfeasibility evaluation.

The test data generation attempts to execute selected infeasible paths resulted, in all the cases, in the correct identification of unfeasibility by the proposed heuristics. This was an expected result since infeasible constraints always lead to the lack of progress of the search.

In the proposed approach the high effectiveness of the Genetic Algorithm's search and the precision of the fitness function have made evident the strong correlation between the persistent lack of search progress and the unfeasibility of the intended path.

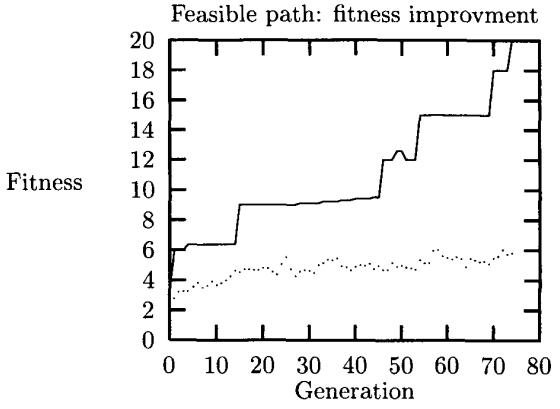

Figure 2: Search dynamics on a feasible path (solid line: best fitness; dotted line: average fitness)

During the entire search the Genetic Algorithm has the precise definition of the "right direction" to improve the test data; consequently, absence of improvement means very often "impossibility of improvement".

As remarks Goldeberg [10], unlike other methods that search the space with a single point, Genetic Algorithms search globally the problem space with a population of points. In the test data generation problem this makes Genetic Algorithms less prone to local-optimal points that could cause the lack of progress in the search for solving (feasible) predicates.

The correct setting of heuristics' parameters (δ, Δ, $Aj1$ e $Aj2$) allows a safe detection of the lack of progress. Previous works using Genetic Algorithms in test data generation do not address the unfeasibility in this way probably because their fitness functions are not precise enough; thus, the persistent lack of search progress happens frequently, even for feasible paths.

Figure 2 illustrates the search dynamics in the generation of test data for a feasible path in the program *expint*. The best fitness increases up to generation 74, where the algorithm finds input data that execute the path. Figure 3 illustrates the search dynamics in the generation of test data for an infeasible path in the same program. The search proceeds regularly up to generation 36 when an infeasible predicate is reached. From generation 36 to 135 the persistent lack of progress occurs (in this case the path has 21 nodes, so $NL = 99$ generations) and path unfeasibility is detected in generation 135.

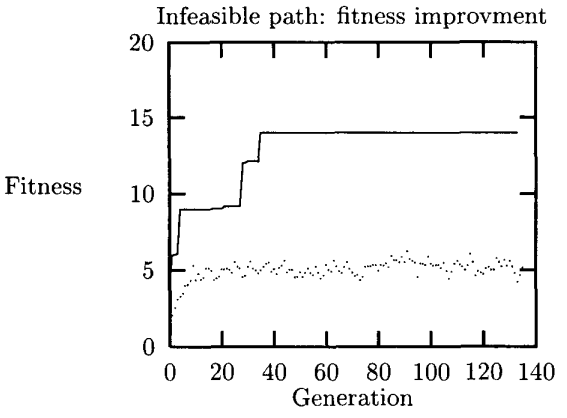

Figure 3: Search dynamics on a infeasible path (solid line: best fitness; dotted line: average fitness)

5.2 Additional remarks on the experiment

As expected, the experiment confirms that:

- The type of predicate the Genetic Algorithm is trying to solve influences the search progress: composed predicates with the logical operator AND; predicates involving equality relational operator (e.g., *if (x == y)*); and predicates with string comparisons (e.g., *if(!strcmp(name, "test1"))*) are harder to solve and tend to generate a higher lack of progress in the search, even when they are feasible. An error rate of 7% happened in paths containing the predicate *if(!strcmp(name, "test1"))*.

- Deeper predicates through the path are harder to solve: the evaluation of previous predicates along the path have to be maintained, despite the changes on input values in the search process. A greater number of previous predicates implies more constraints to keep.

6 Conclusions and future work

An approach is proposed for automation of test data generation and for path unfeasibility identification, tasks that demand from the tester a careful analysis of the program code, a large amount of mental effort and specific knowledge about testing criteria.

The implemented solution integrates well-known techniques: dynamic technique for data generation and optimization using Genetic Algorithms. Compatibility with different variable types and with the diverse structures of language C makes the tool applicable to different types of programs.

This approach to unfeasibility identification differs fundamentally from the previous ones. Gupta et all [12] and Clarke [3] guarantee the unfeasibility identification of paths where the predicates are linear functions of the input variables. The approaches identify all linear infeasible paths and no linear feasible path is identified as infeasible. Our approach, on the other hand, is applicable indistinctly, even to non linear paths.

Our initial evaluation of the proposed heuristics shows that all the infeasible paths are identified and evaluation errors (feasible paths wrongly identified as infeasible) are not frequent, suggesting that it can be useful to structural testing practice.

Improvements on the genetic algorithm and on the heuristics may significantly decrease the computational cost of unfeasibility identification, which is the weak point of this approach concerning its application to real problems.

Contributions of this work include: definition of a new approach to unfeasibility detection using a dynamic heuristics and the definition of a fitness function more precise than previous ones for structural software testing using Genetic Algorithms.

Extensions to the approach may improve this work:

- Development of a more detailed model of complexity for test data generation to provide for a better adjustment of the heuristics. For instance, types of predicates which are more difficult to solve could be identified and taken into account by assigning them higher levels of lack of progress to decide on the unfeasibility of a path.

- Fine tuning the Genetic Algorithm to the problem of test data generation. For instance, approaches adopted in constraint optimization problems [27], similar to the problem of test data generation, could be incorporated in our basic Genetic Algorithm [10] to inprove its performance.

7 Acknowledgements

This work was partially supported by FAPESP and CAPES, and conducted at DCA/FEEC/Unicamp. The authors thank the contributions of: Ivan Ricarte (Unicamp), José Carlos Maldonado (USPSC), the members of DCA/FEEC/Unicamp Test Group and the anonymous referees.

References

[1] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *Software Engineering Notes*, Vol. No 22(6):361–377, November 1997.

[2] M.L. Chaim. *POKE-TOOL - Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise de Fluxo de Dados.* Dissertação de Mestrado, DCA/FEEC/Unicamp, Campinas - SP, Brazil, April 1991. (in Portuguese).

[3] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, Vol. SE-2(3):215–222, September 1976.

[4] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[5] I. Forgács and A. Bertolino. Feasible test path selection by principal slicing. *Software Engineering Notes*, Vol. No 22(6):378–394, November 1997.

[6] F.G. Frankl. *The use of Data Flow Information for the Selection and Evaluation of Software Test Data.* PhD Thesis, Department of Computer Science, New York University, New York, U.S.A., October 1987.

[7] F.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, Vol. SE-14(10):1483–1498, October 1988.

[8] M. J. Gallagher and V. L. Narasimhan. Adtest: A test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, Vol. 23(8):473–484, August 1997.

[9] A.T.C. Goldberg, T.C. Wang, and D. Zimmerman. Applications of feasible path analysis to program testing. In *International Symposium on Software Testing and Analysis*, pages 80–94. ACM, Washington - USA, October 1994.

[10] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Publishing Company, Inc, University of Alabama, 1989.

[11] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. *Foundations of Software Engineering*, Vol. No 11:231–244, November 1998.

[12] N. Gupta, A. P. Mathur, and M. L. Soffa. Una based iterative test data generation and its evaluation. *Proceedings of the IEEE Automated Software Engineering Conference,* October 1999.

[13] D. Hedley and M.A. Hennell. The causes and effects of infeasible paths in computer programs. In *Proceedings of 8th. ICSE*, pages 259–266. UK, 1985.

[14] W.E. Howden. Symbolic testing and dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, Vol. SE-3(4):266–278, July 1977.

[15] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, September 1996.

[16] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, Vol. SE-16(8):870–879, August 1990.

[17] 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 International Conference of The Chilean Computer Science Society.* Santiago, Chile, October 1992.

[18] N. Malevris, D.F. Yates, and A. Veevers. Predictive metric for likely feasibility of program paths. *Information and Software Technology*, Vol. 32(2):115–118, March 1990.

[19] C. C. Michael, G. E. McGraw, M. A. Schatz, and Walton C. C. *Genetic Algorithms for Dynamic Test-Data Generation.* Technical Report RSTR-003-97-11, RST Corporation, May 1997.

[20] R. P. Pargas, Harrold M. J., and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9:263–282, September 1999. to appear.

[21] L.M. Peres, S.R. Vergilio, M. Jino, and J.C. Maldonado. Path selection strategies in the context of software testing criteria. In *1st IEEE Latin American Test Workshop*, pages 222–227. IEEE, Rio de Janeiro, January 2000.

[22] C.V. Ramamoorthy, F. H. Siu-Bun, and W.T. Chen. On automated generation of program test data. *IEEE Transactions on Software Engineering*, Vol. SE-2(4):293–300, December 1976.

[23] 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.

[24] M. Srinivas and Patnaik L. M. Genetic algorithms: A survey. *IEEE Computer*, 27(6):17–26, July 1994.

[25] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings of the ACM Sigsoft International Symposium on Software Testing and Analysis*, pages 73–81. ACM, Florida, USA, March 1998.

[26] S.R. Vergilio, J.C. Maldonado, and M. Jino. Infeasible paths within the context of data flow based criteria. In *VI International Conference on Software Quality*, pages 310–321. Ottawa-Canada, October 1996.

[27] Michalewicz Zbigniew. *Genetic Algorithms + Data Strutures = Evolution Programs.* Springer Verlag, North Carolina - USA, 3rd edition, 1996.