Information and Software Technology 56 (2014) 395–407

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Dynamic stopping criteria for search-based test data generation for path testing

I. Hermadi [a,b,*], C. Lokan [b], R. Sarker [b]

[a] Department of Computer Science, Bogor Agricultural University, Indonesia
[b] School of Engineering and Information Technology, UNSW Canberra, Canberra, ACT, Australia

ARTICLE INFO

Article history:
Received 30 October 2012
Received in revised form 14 December 2013
Accepted 2 January 2014
Available online 10 January 2014

Keywords:
Path testing
Evolutionary algorithm
Software reliability growth model

ABSTRACT

Context: Evolutionary algorithms have proved to be successful for generating test data for path coverage testing. However in this approach, the set of target paths to be covered may include some that are infeasible. It is impossible to find test data to cover those paths. Rather than searching indefinitely, or until a fixed limit of generations is reached, it would be desirable to stop searching as soon it seems likely that feasible paths have been covered and all remaining un-covered target paths are infeasible.
Objective: The objective is to develop criteria to halt the evolutionary test data generation process as soon as it seems not worth continuing, without compromising testing confidence level.
Method: Drawing on software reliability growth models as an analogy, this paper proposes and evaluates a method for determining when it is no longer worthwhile to continue searching for test data to cover un-covered target paths. We outline the method, its key parameters, and how it can be used as the basis for different decision rules for early termination of a search. Twenty-one test programs from the SBSE path testing literature are used to evaluate the method.
Results: Compared to searching for a standard number of generations, an average of 30–75% of total computation was avoided in test programs with infeasible paths, and no feasible paths were missed due to early termination. The extra computation in programs with no infeasible paths was negligible.
Conclusions: The method is effective and efficient. It avoids the need to specify a limit on the number of generations for searching. It can help to overcome problems caused by infeasible paths in search-based test data generation for path testing.

1. Introduction

Testing is a critical but expensive part of the software development life cycle [1–3]. There is considerable interest in ways to automate testing, to reduce the cost and to gain more confidence in the result [4].

A major task in software testing is test data generation [3]. Search-based test data generation aims to automate this task, by searching for test cases (inputs, or pairs of input–output) that satisfy chosen testing criteria.

Most research in this area considers "white box" testing, or structural coverage, in which the aim is to ensure that executing a collection of test cases results in all parts of a program being tested. This can be interpreted in various ways, including "statement coverage" (when the program is tested with all of the test cases, somewhere along the line every statement in the program is executed at least once), "branch coverage" (both outcomes at every logical branch in the program are executed at least once), and "path coverage" (every distinct path through the code is executed at least once). Path coverage is the strongest form of structural coverage [5]. This paper considers path coverage.

Many approaches have been used in path testing [2,6]. Evolutionary path testing, which uses an evolutionary algorithm (e.g. genetic algorithm "GA") as the search engine has been found effective [6,7]. In this research, GA is used as the search engine.

A challenge for any search-based approach is deciding when to terminate the search. If everything sought has been found, the search can stop, but what if not every target has been found yet? Further searching might find more targets, or it might be fruitless because the remaining targets are infeasible (cannot ever be found). If we do not know whether unfound targets are feasible or not, there is a tradeoff between possibly wasting effort searching for something infeasible and possibly missing something feasible by stopping the search too early. In general, an evolutionary algorithm requires one or more stopping criteria to halt the evolution [8]. So far, the most widely used criteria are objective value, fitness

* Corresponding author at: Department of Computer Science, Bogor Agricultural University, Indonesia. Tel.: +62 8111174974.
E-mail addresses: i.hermadi@student.adfa.edu.au, irmanhermadi@apps.ipb.ac.id (I. Hermadi), c.lokan@adfa.edu.au (C. Lokan), r.sarker@adfa.edu.au (R. Sarker).

0950-5849/$ - see front matter © 2014 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2014.01.001

value, number of generations, time elapsed, number of stall generations, and stall time.

In path testing, some target paths may be infeasible. An infeasible path is one for which there is no test data that will cause it to be executed. Searching for data to cover such a path can never succeed. Some criterion is needed to stop the search, to save worthless searching. This means being confident that further searching is indeed worthless: that every path that has not been covered yet is infeasible.

Some stopping criteria used in path testing are path achievement [9], number of generations [10–19], and convergence (lack of fitness improvement) [2,4,20]. Number of generations is the most popular stopping criterion for path testing, followed by convergence. Deciding the maximum number of generations means defining the value of a parameter. This is more a trial and error process than an analytic one. The convergence criterion monitors the best fitness value improvements for certain number of generations. This involves two parameters: improvement size, and the number of stall (or monitored) generations. For example, Mansour and Salame [2] watched for no best fitness improvement for consecutive 50 generations in order to stop the search. Bueno and Jino [4,20] used the convergence stopping criterion in conjunction with infeasible path detection.

The research objective is to find one or more criteria that can be used to decide to terminate the evolutionary test data generation process as soon as it is not worth continuing, without lessening the testing confidence level. This can be achieved if there is a mechanism that can predict the likelihood that the generator will produce new test data that can cover an as-yet-uncovered path in the next generations. Searching can stop when that likelihood becomes low, suggesting that paths that have not been covered yet in the search process are most likely infeasible.

The method proposed in this paper was inspired by reliability growth models that are used in software testing [21]. These models can predict the testing time required to achieve given reliability, or the reliability achieved after a certain testing duration has elapsed.

In software reliability growth modeling, reliability means the probability of failure-free operation of software at a particular time in a certain environment; in other words, the probability that further testing will not expose new bugs. When this reaches chosen levels, testing can stop. There is a risk that undiscovered defects remain, but the risk is judged to be low. Analogously, in search-based test data generation, it can be interpreted as the probability that further searching will not find test data that covers new target paths. There is a risk that test data could still be found to cover uncovered paths, but the risk is judged to be low.

The stopping criterion we propose and evaluate is to halt the GA at the point where it is judged with high confidence that the probability of finding test data to cover a new path in the next generation of searching is less than some threshold.

This paper is organized as follows: Section 3 describes related work. Section 2 presents some theoretical background to understanding this research, and Section 4 describes the proposed approach in detail. Section 5 describes the test programs, and the experimental design and setup. Section 6 presents the results. The results are analysed and discussed in Section 7. Threats to validity are discussed in Section 8. Section 9 concludes the paper.

2. Background

2.1. Path testing

The objective of path testing is to search for a collection of test cases (inputs to a program) that between them lead to the traversal of all logical paths through the program.

In general, path testing process consists of two major steps: target paths generation, and test data generation.

2.1.1. Target paths generation

Target paths generation means identifying a set of logical execution pathways through the program, that we hope should all be exercised during testing.

The source code is needed to construct its logical control flow, which can be presented in a control flow graph (CFG). This graph can be automatically generated by using appropriate programming language grammar in which the program is written.

From the CFG, the different logical paths through the program need to be enumerated. A logical path is a particular flow of execution through the program, which is determined by the decisions made at each decision point between the program's entry point and its exit point.

For a program without loops, the number of logical paths is equal to its cyclomatic complexity (CC) number, or number of basic paths.

The presence of loops can increase the number of logical paths greatly. Each different number of iterations of a loop (e.g. zero: the loop is not executed at all; one: the loop is executed only once before its termination condition is met; two: the loop is executed twice before its termination condition is met) is considered to be a different logical path. For many loops the possible number of iterations is unknown. Target paths should include zero iterations, one iteration, and multiple iterations.

In order to make testing practical, in this research we limit the number of iterations to zero, one and two iterations. Thus a program that has a single loop will have at least 3 target paths; not entering the loop, entering the loop once, and repeating the loop twice. Even with this limited version of path testing, the presence of multiple loops greatly increases the number of target paths. Two loops in sequence have 9 distinct logical paths (3×3 repetitions). Nested loops have 7 distinct paths ($1 + (2 \times 3)$ repetitions).

2.1.2. Test data generation

Generating test data that fulfill path coverage is the main task in path testing. It is the process of creating test data, either heuristically or randomly. In a heuristic approach, the process is guided by some rules to search for required test data; the alternative is that random test data is generated.

2.2. Evolutionary path testing

Path testing that uses any methods from the evolutionary algorithms family is called evolutionary path testing.

In this work, genetic algorithm (GA) is used as the test data generator. A chromosome represents one set of test data (a collection of input values that represents a single test case). Thus the population is a collection of test cases. Each test case causes one target path to be executed; most of the time a target path can be covered by many test cases. The aim is to evolve a set of test cases that causes all target paths to be executed.

Generic steps in GA are (1) Initialization, (2) Evaluation, and (3) Do the following until any stopping criteria is met: (3.a) Selection, (3.b) Perturbation, and (3.c) Go back to Step (2). Initialization generates the first population, randomly or with some knowledge. Step (2) evaluates all members of the population using a given fitness function. In (3.a) some members of the population are selected for perturbation using genetic operators. Section (3.b) applies those operators: crossover is responsible for mixing the genetic traits, and mutation for introducing new genetic traits.

The generator keeps a list of target paths that have not yet been covered. At the beginning of the evolution, every target path is in that list. In each generation, each test case in the population is

evaluated (its fitness is calculated) against each uncovered target path. When a test case is found to cover a target path, it is remembered and that target path is removed from the list. As the search progresses, the list of paths for which test data is sought changes dynamically. Searching can stop if the list becomes empty, or when some other stopping criterion is reached. If the list of target paths contains infeasible paths, the list of uncovered paths will never be empty, and another stopping criterion is essential.

2.3. Software reliability modeling

Software reliability growth models are designed to predict the expected number of failures (failure intensity) at a certain point in time during software testing. The calculation is based on the history of failures occurring during the testing so far.

The model used in this research is based on the Logarithmic Poisson Execution Time Model proposed by Musa and Okumoto in 1984 [21].

The following equations are the model basic forms [21].

$$\mu(\tau) = \frac{1}{\theta} ln(\lambda_0 \theta \tau + 1) \quad (1)$$

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \quad (2)$$

$\mu(\tau)$ represents the number of failures occurred during execution time τ. $\lambda(\tau)$ is the failure intensity, the expected number of failures at a certain point in time. λ_0 is the initial failure intensity. θ is the rate of reduction in the normalized failure intensity per failure.

In [21], the two unknown parameters λ_0 and θ are estimated using the maximum likelihood estimation method to guess the product of $\Phi = \lambda_0 \theta$ by using conditional joint density function. Two types of failure data were used for the estimations: failure intervals and number of failures per interval. Assume that an observation interval $(0, x_p]$ is partitioned into a set of p disjoint subintervals $(0, x_1], (x_1, x_2], \ldots, (x_{p-1}, x_p]$ and the number of failures in each subinterval is recorded. Let $y_l (l = 1, 2, \ldots, p)$ be the number of failures in $(0, x_l]$. So,

$$\hat{\theta} = \frac{1}{y_p} ln\left(\hat{\Phi} x_p + 1\right) \quad (3)$$

and

$$\hat{\lambda_0} = \frac{\hat{\Phi}}{\hat{\theta}} \quad (4)$$

Three assumptions are made in the Logarithmic Poisson model [21]. We observe that they correspond well to the situation of search based path testing with GA. The first assumption is that there is no failure at time $\tau = 0$; this corresponds to no paths being covered when the test data generation is yet to start. The second assumption is that the failure intensity will decrease exponentially with the expected number of failures experienced; this corresponds to the probability of covering a new path in each generation of searching decreasing exponentially as the search progresses. The third assumption is that for a time interval $\Delta\tau$ the probability of more than one failure during $(\tau, \tau + \Delta\tau]$ tends to zero as the time interval tends to zero; this underpins the model being a Poisson-type model.

3. Related work

The following describes related work on metaheuristic based test data generation, input domain reduction, length of test cases, and automated testing tools. McMinn et al. [3] conducted a survey on search based software test data generation in 2004. The survey covers the use of metaheuristic search techniques for generating test data and presents the direction of research in the area.

In 1994, Pei et al. [9] were inspired by the lack of efficacy and efficiency in the search methods used for generating test data using actual program execution. They developed a path-coverage test data generator that employs a genetic algorithm, using two proposed fitness functions, i.e. considering number of matching branches vs. summing of all the branch functions. A minimum–maximum program was used for the test program, whose output is the minimum and maximum numbers in an array of integer numbers. They found that 8 out of 21 selected target paths were infeasible, and showed that their approach could find all feasible target paths.

In 2000, Lin and Yeh [10] extended Jones et al.'s work in 1996 [22]. Jones et al. had worked on branch testing using weighted Hamming distance. Lin and Yeh increased the level of coverage from branch testing to path testing, and extended the ordinary (weighted) Hamming distance so that it can handle different ordering of the target paths that have the same branch nodes. They used the triangle classifier as the test program and reported that the quality of generated test data is higher than the ones produced by random testing.

In the same year, Bueno and Jino [4] proposed an approach that used control and data flow dynamic information to fulfill path coverage testing. The number of coincidence nodes between a target path and the actual (or executed) path was used as a similarity metric, that is summed up with normalized branch distance predicate in nodes where the actual path deviated from the target, to make up the fitness function. In addition, they also tackled the identification of potentially infeasible paths by monitoring the progress of the search for required test data. Six test programs were used. They continuing the work on evolutionary path testing in 2001 [23] and 2002 [20], with similar fitness function.

In 2003, Hermadi and Ahmed [12] presented evolutionary path testing using multiple paths, i.e. attacking all target paths at once instead of seeking to cover a single path in each search. They proposed and investigated different fitness functions, that are composed of combinations of approximation level and branch distance (similar to Bueno and Jino's work [4]). Things considered in forming the fitness functions were path traversal technique, neighborhood influence, weighting, and normalization. Three test problems were used. The work suggested promising results, and a particular fitness function was identified that outperformed the others (that function is used in this research; see Section 5.3 below).

One year later, Mansour and Salame [2] compared two algorithms to generate data for path testing, i.e. Simulated Annealing (SA) and Genetic Algorithm (GA). Their fitness function was made of weighted Hamming distance between the operands of each predicate of corresponding nodes from the target path and the actual (or executed) path. Eight test problems were exercised to validate the approach, with cyclomatic complexity numbers ranging between 2 and 14. The empirical results showed that SA tended to outperform GA slightly.

In 2008, Ahmed and Hermadi [15] extended Hermadi and Ahmed's work in 2003 [12]. A rewarding scheme was introduced and more efficient multiple path evolutionary path testing was developed.

In the same year, Chen and Zhong [16] proposed multi-population GA (MPGA) for path testing. The fitness function was the summation of all branch predicate distances between a target path and the actual path. Triangle classifier was the only test program used to validate the approach. The work confirmed empirically that MPGA is more effective and efficient than the ordinary single population one.

In 2006, McMinn et al. [24] proposed Species per Path approach to factor out target paths in path coverage testing, such that searching for infeasible paths could be avoided. This means each

path has its own evolutionary process that can be executed in parallel. The experimental results show the approach is effective and efficient.

In 2007, Harman et al. [25] proposed an approach to dynamically reduce the input domain size using symbolic execution. The input is fed into the program and executed symbolically, and then the constraints in the branches are traced back to the input domain so that its size can be reduced. The approach is effective in reducing the input domain size, although it is time consuming work.

In 2010, Lakhotia et al. [26] developed a test data generator AUSTIN to satisfy branch coverage for C programs. The generator uses an evolutionary test framework. They validated the proposed approach on deployed automotive systems. The test results show that the approach is effective and efficient in generating test data.

In 2011, Arcuri [27] worked on the length of test cases to achieve branch coverage in object-oriented programs. A test case is a sequence of method calls. In his study, the length of test cases can grow easily and, intuitively, longer test cases can cover more branches, i.e. achieve higher branch coverage. Arcuri proposed several techniques that can control its growth. The experimental results, which made use of the EVOSUITE tool [28], show that the methods can significantly improve the test case generator performance. Further, Arcuri added that longer test cases made difficult test programs easier to test [27].

4. Proposed approach

In general, GA has several stopping criteria that can be used to end its evolutionary process.

In path testing, it is quite likely that some target paths are infeasible, and no test data can lead to execution of these paths. Having one or more of these paths in the set of target paths will mean that GA can never stop as a result of having covered all target paths; other stopping criteria must be used to stop the search.

We propose that the Logarithmic Poisson execution time model can be used to predict the probability of finding new path(s) within a generation of GA based path testing. This is analogous to predicting failure intensity in a given time interval in modeling software reliability growth: searching for one more generation is analogous to testing for one more time interval.

This probability can be employed as a new stopping criteria for GA. The search can be terminated when the probability of finding test cases to cover new paths falls below a threshold; at that point, it is assumed that further searching is unlikely to cover more paths, and that any paths that remain uncovered are infeasible. By setting the threshold high or low, one can trade off searching time against the confidence that all feasible paths have been covered.

To apply this idea, at a given generation number τ we need to calculate the failure intensity $\lambda(\tau)$ from Eq. 2. This means we must know the values for λ_0 and θ. For λ_0 we use the number of paths found in the first generation of searching; θ is found by fitting a curve of the form of Eq. (2), based on the history of how many new paths were covered in each generation so far. As the search progresses and finds new paths, θ estimation will be refined.

For example, suppose that in 25 generations of searching, we record that 13 paths were covered in the first generation; no new paths were covered in the second, third or fourth generations; two further paths were covered in the fifth generation; no more until the sixteenth generation when one further path was covered; no more until the twenty-second generation when one further path was covered; and no more after that. This history can be represented as a series of tuples (x, y), where x represents the generation number and y represents the number of paths newly covered in that generation. A new tuple is added to the history with each generation of searching.

After two generations this history can be represented as {(1,13), (2,0)}; after five generations it would be represented as {(1,13), (2,0), (3,0), (4,0), (5,2)}; and so on.

After the second generation, the first two points can be used to fit a curve of the form of Eq. (2); this gives the top line in Fig. 1. An initial value is needed for θ: based on experimentation, we use 0.3.

After each generation, the curve can be re-fitted. Fig. 1 shows the fitted curves, using this example data, obtained after each generation from the second to the twenty-fifth. The top curve is the roughest one based on the fewest pairs of data. The bottom curve is the finest one based on all 25 tuples.

As more data becomes available, the curves tend to converge. In this example, the curve becomes quite stable in fewer than 10 generations.

After each generation, the fitted equation can be used to estimate the probability of covering a new path at each number of generations in the future. This can be re-cast as estimating the probability of covering a new path in the next generation of searching, allowing a decision to be made about when to stop searching.

4.1. Decision rules

One of the main research questions is how to make use of the model to identify when to stop the evolutionary path testing because it is not useful to continue the search anymore.

There are two elements to this: the probability of covering new paths falling below a threshold ("reliability rule") and the stability of the equation that predicts that probability ("stability rule").

The reliability rule can be used to decide to stop the search when the probability of finding new paths $\lambda(\tau)$ in generation τ, as given by Eq. (2), is small enough. This rule is based on the value of $\lambda(\tau)$ at generation τ.

Stability of the reliability equation is also important. If the equation used to predict when to stop searching is still uncertain enough (i.e. the estimate of θ has not yet converged), it could be premature to decide to stop searching based on the current estimate of $\lambda(\tau)$. This rule is based on $\Delta\theta$, the change in the estimate of θ at generation τ.

In the experiments that follow we investigate each rule separately, and in combination.

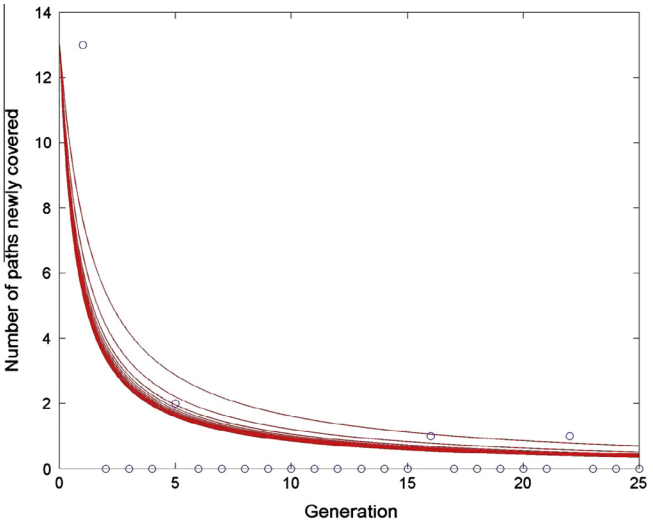

Fig. 1. Plot of $\lambda(\tau)$.

4.2. Evaluating decision rules

The performance of a rule is assessed based on how many feasible paths are missed if the search stops at the number of generations suggested by the rule, and on searching time.

The number of missed paths is considered in both absolute and relative terms: absolute meaning how many were missed; relative meaning how many were missed when using the rule, compared to how many were missed if the rule was not used and instead the search stopped after a fixed number of generations.

Searching time is evaluated in terms of the number of generations of search when the rule was used, compared to the number of generations of search when the rule was not used and instead the search stopped after a fixed number of generations.

5. Experiments

The purpose of the experiments is to study the feasibility of applying a reliability growth model as a stopping criterion in evolutionary path testing. We compare it against the common approach of searching for a fixed number of generations. The aim is to ascertain how much is gained or lost, in execution time and path coverage. The considerations are overhead due to the proposed method (computation time to apply it), searching time (how much time is saved or lost, compared to searching for a standard number of generations?) and searching effectiveness (how many feasible paths remain uncovered using this stopping condition, compared to searching for a standard number of generations?).

Other questions that have been driving forces for this work are: What kind of information that is helpful can be extracted from the model? What confidence level can be achieved should the approach be applied? Can the model reliability be maintained across different runs for same test program? Or across different test programs?

This section describes our experimental setup: selection of test programs; choosing which test programs to use for tuning (to identify suitable values for thresholds in the decision rules) and testing; use of GA (fitness function, parameters setup, number of runs); and the decision rules that we investigated.

5.1. Test programs

Twenty-one programs (written in C language) were used in the experiments. They were gathered from the literature, and have all been used as test problems for research in test data generation in search based software engineering (SBSE) [29]. All but three are complete programs. The others (bG2011, fG2011, sG2011) are portions of larger programs; we used the same portion as others have used in past research in this field [30].

Between them these programs cover many elements of computer programs, e.g. no loop, single loop, many loops, loops in sequence, nested loops, simple selection statements, and complex selection statements with combinations of arithmetic and/or relational operators.

Table 1 presents characteristics of the test programs: number of lines of code **LOC**, extended **CC** [31,32], number of loops **L**, number of branches **B** (this can be either selection or loop statements, e.g. **IF, IF-ELSE, WHILE, FOR, DO**), branching structure **Structure**, total number of target paths **P** including both feasible and infeasible paths, number of feasible paths **F**, and its reference(s) **Refs**. **Structure** is represented as prefix statement with the following conventions: serial **-** and nested/inside **()**. For example, Fig. 2 (which represents the Triangle test program) has 3 branches, all of which are **IF** statements; it is represented as **IF-(IF-ELSE-(IF-ELSE))-ELSE** in Table 1. The explanation is that the inner most **IF-ELSE** statement is inside the **ELSE** part of the second inner **IF-ELSE** statement and this second inner **IF-ELSE** is inside the **IF** part of the outer most **IF-ELSE** statement.

The following are brief descriptions for each test program:

- mmA2008 finds the minimum and maximum values from an array of integers.
- iA2008 sorts an array of numbers in non-decreasing order using insertion sort method.
- bisA2008 finds the root(s) of non-linear equation using bisection method.
- mtA2008 consists of Minimaxi and Triangle in sequence.
- tA2008 determines whether three given numbers that represent three lengths on a plane form a scalene, isosceles, equilateral, or not a triangle.
- binA2008 searches a key item in an array of numbers using binary method by returning its index number if the key is found and null otherwise.
- bubA2008 sorts an array of numbers increasingly using bubble sort method.
- gA2008 finds the greatest common divisor between any given two integers.
- rA2008 finds the remainder in integer division.
- tM2004 (Triangle) classifies three numbers representing triangle side lengths into five type triangles: scalene, isosceles, right, iso-right, or equilateral.
- erR1985 (Expint) raises one integer to the power of the other.
- qG1997 (Quotient) calculates the quotient and the remainder of the division of two positive integers.
- ttB2002 (Tritype) accepts three integers representing sides of a triangle, classifies its type, and computes its area.
- eiB2002 (Expint) accepts an integer and a float variable for exponentiation.
- qB2002 (Quotient) is similar to QG1997 with a slightly different process.
- scB2002 (Strcomp) compares three characters and a string with five positions.
- fcB2002 (Floatcomp) compares three floating point numbers and has some selections.
- fB2002 (Find) seeks for a certain key in an array.
- bG2011 (Bubble) sorts an array of numbers using bubble sort.
- fG2011 (Flex) is a unix utility program that is reconstructed from [30].
- sG2011 (Space) is a program that reads a file that contains several ADL statements and check the contents of the file for adherence to the ADL grammar and to specific consistency rules is provided in [30].

5.2. Tuning and testing programs

We divided the test programs into two groups, for tuning and testing. We used the tuning group to determine suitable values for thresholds for $\lambda(\tau)$ and θ in the decision rules. We used the testing group to evaluate the resulting rules. Each group was selected based on program's characteristics, complexity and target paths feasibility.

Four programs were selected for tuning. These are the first four programs in Table 1: mmA2008, iA2008, bisA2008, and mtA2008. These four programs cover a range of different program characteristics, e.g. two include infeasible paths, there is a wide range of CC, and all include loops.

5.3. Fitness function

A path representation needs to be described first in order to understand the fitness function. A target path is represented as a

Table 1
Test programs (in C language).

Program	LOC	CC	L	B	Structure	P	F	Refs.
mmA2008	12	[4: 4]	1	3	WHILE–(IF–IF)	13	13	[9,12,15]
iA2008	15	[3: 4]	2	2	FOR–(WHILE)	6	5	[15,33]
bisA2008	27	[5: 7]	1	4	IF–WHILE–(IF–ELSE–(IF–ELSE))	9	6	[15,34–36]
mtA2008	31	[7:16]	1	6	WHILE–(IF–IF)–IF–(IF–ELSE–(IF–ELSE))–ELSE	52	20	[12,15]
tA2008	14	[4:12]	0	3	IF–(IF–ELSE–(IF–ELSE))–ELSE	4	4	[10,12,15,16,22,33,35,37–41]
binA2008	22	[3: 3]	1	2	WHILE–(IF–ELSE)	7	7	[15,22]
bubA2008	18	[4: 5]	2	3	WHILE–(FOR–(IF))	15	4	[15,35]
gA2008	16	[4: 4]	1	3	IF–ELSE–(WHILE–(IF–ELSE))	8	5	[15,33]
rA2008	16	[4: 4]	1	3	IF–ELSE–(IF–ELSE–(WHILE))	5	4	[15,22,36,41]
tM2004	20	[5: 5]	0	4	IF–(IF–ELSE)–ELSE–(IF)–IF	8	7	[20]
eR1985	18	[4: 4]	1	3	IF–ELSE–WHILE–IF	12	5	[42]
qG1997	16	[4: 4]	2	3	WHILE–WHILE–(IF)	21	4	[43]
ttB2002	38	[8:11]	0	7	IF–ELSE–(IF–ELSE–(IF–(IF–ELSE–(IF–ELSE)))–ELSE–(IF–ELSE–(IF–ELSE))	8	8	[20]
eiB2002	90	[11: 15]	3	11	IF–ELSE–(IF–ELSE–(IF–ELSE–(IF–(FOR–(IF))–ELSE–(IF–FOR–(IF–(IF)–ELSE–(IF))))))	31	5	[4,20,23,44]
qB2002	23	[6: 7]	2	5	IF–(IF–((WHILE–WHILE–(IF))))	27	10	[4,20,23]
scB2002	17	[6: 7]	1	5	WHILE–IF–(IF–(IF–(IF)))	16	4	[4,20,23]
fcB2002	21	[5: 8]	0	4	IF–(IF–(IF–(IF)–ELSE))	5	5	[4,20,23]
fB2002	70	[9: 10]	3	8	WHILE–(–IF–IF–(IF–(IF))–ELSE–(WHILE–WHILE–IF))	32	8	[4,23,20]
bG2011	37	[4: 4]	2	3	FOR–(FOR–(IF–ELSE))	20	11	[30]
fG2011	95	[7: 7]	0	6	IF–(IF–IF)–IF–(IF)–IF	30	30	[30,45]
sG2011	72	[6: 6]	0	5	IF–IF–IF–IF–IF–IF	32	32	[30,45]

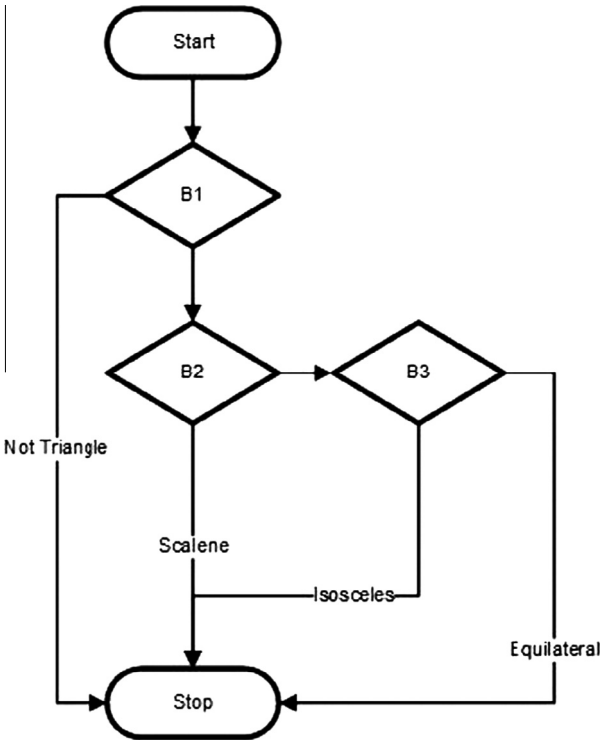

Fig. 2. Triangle's CFG.

sequence of pairs, of predicate (or branch) number, e.g. B1 for first branch, and its decision (1 or 0 for TRUE or FALSE respectively). If there are repetitions of sub path(s) in a path then they indicate presence of loops.

For example, one of the paths in Triangle (see Fig. 2) can be written as the sequence (1 1 2 0 3 1), which means the TRUE branch is taken at B1, the FALSE branch at B2, and the TRUE branch at B3. This is the path taken with data that represents an isosceles triangle. In Fig. 2, each type of triangle corresponds to a different path, giving 4 paths in total. These paths are Not Triangle (1 0), Scalene (1 1 2 1), Isosceles (1 1 2 0 3 1), and Equilateral (1 1 2 0 3 0).

The fitness function used in the experiments consists of approximation level (AL) and branch distance (BD) [15]. AL measures similarity (or dissimilarity) between the path taken by an input data and the target path; it is counted as the number of matched (or unmatched) branches. The count continues as far as the first unmatched branch encountered. BD is calculated as Korel's distance function [46], if the path taken differs from the target path of interest.

For example, see Fig. 2, given a target path (1 1 2 1), test data (8, 3, 8) corresponding to (Input1, Input2, Input3), and the predicate at B2 (Input1 $\leqslant$ Input2). The path taken is (1 1 2 0 3 1), so AL is 1, because they only traverse the same branch at B1. In this case, branch distance is calculated at B2 with Korel's distance function [46], i.e. (Input1 − Input2) + 1, which equals 6.

AL and BD can be combined in different ways, each combination representing a different fitness function. 32 different fitness functions were studied in [47], with one found to be the best: considering all current target paths (or multiple target paths), path-wise traversal, normalization (distance, violation, and fitness value), and no weighting for AL and BD. That fitness function is used here, as described in Eq. (5).

$$F_j = \frac{\sum_{i=1}^{|TG|}\sum_{i=1}^{lC_{ij}}(BD_{ij} + AL_{ij})}{|\,TG\,|} \tag{5}$$

where i is the (target) path number, j is the input number, $|\,TG\,|$ is the number of (target) paths, lC_{ij} is the minimum length between the target path i and the traversed path j, BD_{ij} is the normalized branch distance between the target path i and the traversed path j over the maximum branch distance in the current generation, and AL_{ij} is the normalized approximation level between the target path i and the traversed path j over the length of path i.

5.4. Setup

The following GA common parameters are used: random seeded initial population generation, roulette wheel selection, 90% single point crossover, and 90% generation gap.

Two different approaches were used to set the values for other parameters. One was to set the parameters separately for each test program, using the best values for each program as found in [19], as shown in Table 2. The other, to assess generality, was to use common values (population size 100, mutation rate 0.1, number of generations 100) that represent a balance between the program-specific best values.

Table 2
Best parameter settings.

Program	Pop	Gens	Allele	Mut
mmA2008	200	100	−10 10	0.15
iA2008	70	50	−10 10	0.15
bisA2008	100	100	1.72 1.75	0.30
mtA2008	250	50	0 20	0.10
tA2008	150	250	0 10	0.30
binA2008	30	50	−10 10	0.15
bubA2008	50	50	−10 10	0.30
gA2008	250	50	0 20	0.10
rA2008	250	500	0 20	0.10
tM2004	150	250	0 10	0.30
eiR1985	50	50	0 10	0.30
qG1997	50	50	1 10	0.30
ttB2002	250	500	0 20	0.15
eiB2002	250	500	−10 10	0.15
qB2002	250	500	0 20	0.15
scB2002	250	500	1 128	0.15
fcB2002	250	500	−10 10	0.15
fB2002	100	100	1 200	0.15
bG2011	100	50	−10 10	0.15
fG2011	250	500	−10 10	0.15
sG2011	250	50	−10 10	0.15

Allele range depends on the nature of the test program, so cannot be set to standard values.

The purpose of the experiments is to compare the searching performance with the proposed stopping condition against the criterion of stopping after a fixed number of generations. That number was set at 100 generations across all test programs.

30 runs were made with each program and each parameter setup.

5.5. Rules

There are two main tasks in designing the experiments. The first task is finding optimal thresholds for each rule, in terms of effectiveness and efficiency.

- With the reliability rule, the search stops as soon as $\lambda(\tau)$ reaches a threshold. We investigated thresholds of 0.5 (at which point the probability of covering a new path in the next generation is 0.5), 0.25, and 0.1.
- In the stability rule, the controlling parameter θ of reliability curve is investigated. In order to monitor the changes of curve shapes, $\Delta\theta$ (the difference between the current and the previous value) is used. When $\Delta\theta$ reaches a threshold, the search stops. We investigated threshold values of 0.001, 0.0005, 0.00033, 0.00025, 0.0002, and 0.0001.

The second task is investigating the strength and weakness of each rule, and finding the best combination of both.

5.6. Computation time

An important point of implementing the proposed stopping condition is its computational time. In other words, rule execution should not cause computation overhead. If it did, the benefit of saving time from stopping the search much earlier is not significant.

In the experiments, we measured the execution time required to fit the curve of Eq. (2), to calculate $\lambda(\tau)$ and $\Delta\theta$ and to apply the decision rule, at each generation. We also measured the computation time required for each generation of searching.

6. Results

This section presents experimental results to evaluate the efficacy and efficiency of the proposed approach. Efficiency relates to how much execution time is saved or lost from applying the proposed approach, compared to the common approach of searching for a fixed number of generations. Efficacy relates to how many feasible paths are found or missed, compared to searching for a fixed number of generations.

We use **Gens** = 100 for the fixed number of generations. Table 2 shows that the best numbers of generations for the programs studied here range from 50 to 500; we chose 100 generations as a compromise. Clearly the value chosen for **Gens** would affect the results; this is discussed in Section 7.4.

Other abbreviations in the tables and discussion that follow have the following meanings:

- λ: the threshold value for $\lambda(\tau)$ at which a rule suggests to stop searching;
- $\Delta\theta$: the threshold value for $\Delta\theta$ at which a rule suggests to stop searching;
- **GR**: the number of generations at which a rule suggests to stop searching;
- **PF**: the average number of paths found over 30 runs if searching continues for **Gens** generations;
- **PFM**: the maximum number of paths found in any run if searching continues for **Gens** generations;
- **PFR**: the number of paths found over 30 runs if searching continues until the rule suggests to stop;
- **PFRM**: the maximum number of paths found in any run if searching continues until the rule suggests to stop;
- **MP**: the average extra number of paths missed if the search terminates after **GR** generations instead of after **Gens** generations (**MP**): **MP** = **PF** − **PFR**;
- **Eff**: the efficiency achieved, i.e. the percent of search time avoided, compared to searching for **Gens** generations: **Eff** = $(1 - (\mathbf{GR}/\mathbf{Gens})) \times 100$.

6.1. Execution overhead to apply the proposed approach

The first result is that the execution time required by the method is negligible. In our environment, the time required to fit the curve and use it to estimate $\lambda(\tau)$ and $\Delta\theta$ was 0.05 s, with 100 data points. The average execution time for one generation of searching was about 60 s. Thus the effort to apply the method is ignored in the rest of the paper.

6.2. Tuning programs

6.2.1. Reliability Rule (RR)

Table 3 summarizes the performance of the reliability rule (stop searching as soon as $\lambda(\tau)$ falls to a given threshold) for the four tuning programs. At λ = 0.5, the chance of finding an uncovered path in the next generation of searching is 50%; not surprisingly, stopping at this threshold value avoids the most computation (**Eff** averages 88% cross the four tuning programs), but has the greatest chance of missing some feasible paths compared to searching for 100 generations. At $\lambda = 0.1$, there is very little chance that a path is missed, but the efficiency is lower (the average is 40% across the four tuning programs).

From Table 3, $\lambda \leq 0.25$ appears a reasonable trade off between efficiency (**Eff** is above 60% for all four programs) and effectiveness (**MP** is low in all four programs). Based on this, we chose $\lambda \leqslant 0.25$ and $\lambda \leqslant 0.10$ for more thorough testing.

Table 3
Tuning programs, reliability rule.

Program	λ	GR	PFR	MP	Eff. %
mmA2008	0.50	16.00	12.93	0.07	84.00
	0.25	32.23	13.00	0.00	67.77
	0.10	80.94	13.00	0.00	19.06
iA2008	0.50	6.29	5.00	0.00	93.71
	0.25	13.20	5.00	0.00	86.80
	0.10	33.10	5.00	0.00	66.90
bisA2008	0.50	6.51	5.43	0.30	93.49
	0.25	13.39	5.57	0.16	86.61
	0.10	34.04	5.67	0.06	65.96
mtA2008	0.50	18.02	16.62	0.58	81.98
	0.25	36.31	17.13	0.07	63.69
	0.10	91.09	17.20	0.00	8.91

6.2.2. Stability Rule (SR)

Table 4 summarizes the performance of the stability rule (stop searching when the equation has stabilized) for the four tuning programs. While this is not linked directly to the probability of covering new paths, it gives some indication of how much confidence to give to the estimate of **GR**.

$\Delta\theta \leqslant 0.00025$ has good efficiency across the four tuning programs, and is the point at which **MP** falls to zero for three of the four tuning programs. $\Delta\theta \leqslant 0.0001$ is the first point at which **MP** is zero for all tuning projects, but its efficiency is lowest. Based on this, we chose $\Delta\theta \leqslant 0.00025$ and $\Delta\theta \leqslant 0.0001$ for more thorough testing.

6.2.3. Reliability and Stability Rule (RSR)

Based on the results above, we chose two decision rules to evaluate with the testing programs. These rules combine both the reliability rule and the stability rule. One (**RSR1**) uses the higher (less stringent) values for λ and $\Delta\theta$ (i.e. $\lambda \leqslant 0.25$ and $\Delta\theta \leqslant 0.00025$), and the other uses the lower (more stringent) values (i.e. $\lambda \leqslant 0.1$ and $\Delta\theta \leqslant 0.0001$).

6.3. Testing programs

Each of the **RSR**s was executed using the best parameter setup for each test program, and using the common parameter setup. This leads to a total of four combinations: **RSR1** with best parameters (**RSR1-B**), **RSR1** with common parameters (**RSR1-C**), **RSR2** with best parameters (**RSR2-B**), and **RSR2** with common parameters (**RSR2-C**).

6.3.1. Best parameter settings

Table 5 presents the results for **RSR1-B** for all 21 programs.

For programs with no infeasible paths, the search can stop as soon as all target paths are covered. For most such programs, this occurs before both **Gens** and **GR** generations, so efficiency is not relevant. For some programs with no infeasible paths, however, some paths remain uncovered after **Gens** and/or **GR** generations. These are difficult paths to cover. In six test programs (rA2008, mtA2008, tM2004, ttB2002, fB2002, and fG2011), some feasible paths were missed in some runs.

These difficult paths might be found if searching went on for longer, by setting **Gens** to a large value, or by reducing the parameters influencing **GR**, causing the search to continue for longer; or by incorporating knowledge into the search. This is discussed in Section 7.5.

The mean number of missed paths is 0.48; this is almost all due to one test program. The median number of missed paths is zero. Two test programs have negative efficiencies, i.e. mtA2008 and sG2011. A negative efficiency means that the suggested number of generations by the rule is higher than the default 100 generations. On average, the efficiency is 59%.

6.3.2. Common parameter settings

Table 6 shows the results for **RSR1-C**. As expected, in comparison with **RSR1** with best parameters, the number of missed paths is not as good: it increases to 0.84 on average, compared to searching for **Gens** generations. The number of test programs that miss some feasible paths has doubled, from six to ten test programs. They are gA2008, rA2008, mmA2008, tM2004, eiR1985, qB2002, scB2002, fB2002, fG2011, and sG2011. On average, the efficiency is 74%. Efficiency is better than **RSR1-B**, but effectiveness is worse.

Table 7 displays the results for **RSR2-B**. As it uses both the best parameter settings and the most stringent threshold values, this combination would be expected to miss the fewest paths. This is what we see, with the number of missing paths 0.18 on average.

Table 4
Tuning programs, stability rule.

Program	$\Delta\theta$	GR	PFR	MP	Eff. %
mmA2008	0.001	13.74	12.93	0.07	86.26
	0.0005	21.07	12.93	0.07	78.93
	0.00033	26.03	12.93	0.07	73.97
	0.00025	30.03	13.00	0.00	69.97
	0.0002	33.33	13.00	0.00	66.67
	0.0001	43.17	13.00	0.00	56.83
iA2008	0.001	22.41	5.00	0.00	77.59
	0.0005	31.55	5.00	0.00	68.45
	0.00033	39.41	5.00	0.00	60.59
	0.00025	44.69	5.00	0.00	55.31
	0.0002	49.69	5.00	0.00	50.31
	0.0001	70.82	5.00	0.00	29.18
bisA2008	0.001	23.47	5.70	0.03	76.53
	0.0005	34.93	5.70	0.03	65.07
	0.00033	42.07	5.82	0.00	57.93
	0.00025	47.10	5.79	0.00	52.90
	0.0002	53.63	5.80	0.00	46.37
	0.0001	72.50	5.73	0.00	27.50
mtA2008	0.001	12.17	15.67	1.53	87.83
	0.0005	20.58	16.70	0.50	79.42
	0.00033	24.04	16.88	0.32	75.96
	0.00025	27.32	16.82	0.38	72.68
	0.0002	31.76	17.04	0.16	68.24
	0.0001	64.15	17.22	0.00	35.85

Table 5
Testing programs RSR1-B.

Program	PFM	PF	PFMR	PFR	GR	MP	Eff. %
mmA2008	13	13.00	13	13.00	30.00	0.00	70.00
iA2008	5	5.00	5	5.00	23.00	0.00	54.00
bisA2008	6	6.00	6	6.00	20.93	0.00	79.07
mtA2008	20	19.00	20	18.88	78.79	0.12	−57.58
tA2008	4	4.00	4	4.00	25.83	0.00	89.67
binA2008	7	7.00	7	7.00	19.10	0.00	61.80
bubA2008	4	4.00	4	4.00	29.00	0.00	42.00
gA2008	5	5.00	5	5.00	22.10	0.00	55.80
rA2008	4	4.00	4	3.97	26.27	0.03	94.75
tM2004	7	6.23	7	6.21	21.34	0.02	57.32
eiR1985	3	2.97	3	2.97	29.37	0.00	41.26
qG1997	4	4.00	4	4.00	25.00	0.00	50.00
ttB2002	8	8.00	8	7.53	19.11	0.47	96.18
eiB2002	5	5.00	5	5.00	22.00	0.00	95.60
qB2002	10	10.00	10	10.00	27.07	0.00	94.59
scB2002	4	4.00	4	4.00	27.67	0.00	94.47
fcB2002	5	5.00	5	5.00	22.40	0.00	95.52
fB2002	8	7.87	8	7.67	20.61	0.20	79.39
bG2011	11	11.00	11	11.00	27.33	0.00	45.34
fG2011	30	25.13	18	15.83	31.33	9.30	87.47
sG2011	32	32.00	32	32.00	95.40	0.00	−90.80

Table 6
Testing programs RSR1-C.

Program	PFM	PF	PFMR	PFR	GR	MP	Eff. %
mmA2008	13	11.63	11	11.41	32.5	0.22	67.50
iA2008	5	5.00	5	5.00	22.50	0.00	77.50
bisA2008	6	6.00	6	6.00	20.90	0.00	79.10
mtA2008	18	17.53	18	17.53	67.16	0.00	32.84
tA2008	4	3.83	4	3.83	29.53	0.00	70.47
binA2008	7	7.00	7	7.00	19.00	0.00	81.00
bubA2008	4	4.00	4	4.00	25.23	0.00	74.77
gA2008	5	3.57	1	1.00	6.57	2.57	93.43
rA2008	4	3.97	4	3.80	56.80	0.17	43.20
tM2004	6	4.20	6	3.90	37.83	0.30	62.17
eiR1985	3	2.97	1	1.00	6.19	1.97	93.81
qG1997	4	4.00	4	4.00	25.00	0.00	75.00
ttB2002	8	7.03	8	7.03	19.13	0.00	80.87
eiB2002	5	4.90	5	4.90	23.83	0.00	76.17
qB2002	10	8.63	8	8.00	20.00	0.63	80.00
scB2002	4	3.83	1	1.00	7.00	2.83	93.00
fcB2002	5	4.90	5	4.90	25.16	0.00	74.84
fB2002	8	7.87	8	7.67	20.61	0.20	79.39
bG2011	11	11.00	11	11.00	27.62	0.00	72.38
fG2011	13	9.30	12	7.69	26.31	1.61	73.69
sG2011	32	17.77	18	10.69	27.30	7.08	72.70

Table 7
Testing programs RSR2-B.

Program	PFM	PF	PFMR	PFR	GR	MP	Eff. %
mmA2008	13	13.00	13	13.00	76.27	0.00	23.73
iA2008	5	5.00	5	5.00	72.13	0.00	−44.26
bisA2008	6	6.00	6	6.00	64.90	0.00	35.10
mtA2008	20	19.00	20	18.87	112.70	0.13	−125.40
tA2008	4	4.00	4	4.00	51.37	0.00	79.45
binA2008	7	7.00	7	7.00	58.10	0.00	−16.20
bubA2008	4	4.00	4	4.00	88.00	0.00	−76.00
gA2008	5	5.00	5	5.00	70.20	0.00	−40.40
rA2008	4	4.00	4	4.00	55.17	0.00	88.97
tM2004	7	6.23	7	6.23	66.57	0.00	−33.14
eiR1985	3	2.97	3	2.97	89.53	0.00	−79.06
qG1997	4	4.00	4	4.00	32.67	0.00	34.66
ttB2002	8	8.00	8	7.87	58.60	0.13	88.28
eiB2002	5	5.00	5	5.00	70.00	0.00	86.00
qB2002	10	10.00	10	10.00	61.93	0.00	87.61
scB2002	4	4.00	4	4.00	48.52	0.00	90.30
fcB2002	5	5.00	5	5.00	70.83	0.00	85.83
fB2002	8	7.87	8	7.86	61.38	0.01	38.62
bG2011	11	11.00	11	11.00	66.80	0.00	−33.60
fG2011	30	25.13	30	21.61	90.60	3.46	63.76
sG2011	32	32.00	32	32.00	135.37	0.00	−170.74

Table 8
Testing programs RSR2-C.

Program	PFM	PF	PFMR	PFR	GR	MP	Eff. %
mmA2008	13	11.63	13	11.57	63.50	0.07	36.50
iA2008	5	5.00	5	5.00	71.00	0.00	29.00
bisA2008	6	6.00	6	6.00	65.03	0.00	34.97
mtA2008	18	17.53	18	17.53	91.97	0.00	8.03
tA2008	4	3.83	4	3.83	29.53	0.00	70.47
binA2008	7	7.00	7	7.00	59.00	0.00	41.00
bubA2008	4	4.00	4	4.00	41.37	0.00	58.63
gA2008	5	3.57	5	3.18	106.00	0.32	−6.00
rA2008	4	3.97	4	3.97	72.30	0.00	27.70
tM2004	6	4.20	6	4.10	112.83	0.10	−12.83
eiR1985	3	2.97	3	2.40	91.60	0.57	8.40
qG1997	4	4.00	4	4.00	32.67	0.00	67.33
ttB2002	8	7.03	8	7.03	59.53	0.00	40.47
eiB2002	5	4.90	5	4.90	62.57	0.00	37.43
qB2002	10	8.63	10	8.63	56.60	0.00	43.40
scB2002	4	3.83	4	3.82	60.82	0.00	39.18
fcB2002	5	4.90	5	4.90	68.63	0.00	31.37
fB2002	8	7.87	8	7.86	61.38	0.00	38.62
bG2011	11	11.00	11	11.00	66.73	0.00	33.27
fG2011	13	9.30	13	8.76	89.31	0.45	10.69
sG2011	32	17.77	32	15.96	99.23	0.30	0.77

It only has four test programs that miss any feasible paths, i.e. ttB2002, mtA2008, fB2002, and fG2011. In term of efficiency, more test programs were less efficient compared to searching for 100 generations, i.e. 9 test programs. On average, efficiency is 9%.

Table 8 exhibits the results for **RSR2-C**. This rule uses the same stringent parameter settings for the decision rule as **RSR2-B**, but uses common settings rather than program-specific settings for the GA parameters. Eight programs have an **MP** value above zero, but on average this combination misses the fewest paths compared to searching for 100 generations (0.1 paths). On average, efficiency is 32%.

Table 9 shows which test programs have missed feasible paths with which rules and parameter settings. Only one test program (fG2011) missed feasible paths with every combination.

Table 10 shows statistics of rules in terms of **MP**. **MTP** is the number of test programs that have negative efficiency when the stopping rule is applied, i.e. the rule says it is still worthwhile to keep searching beyond **Gens** generations. MTP increases as the rule becomes more stringent, e.g. **MTP** increases from 2 to 9 for **RSR1-B** and **RSR2-B**.

Table 11 presents statistics of rules in term of **Eff**. **RSR1-C** is the most efficient rule with no test program showing any inefficiencies.

Table 12 summarizes **PF** and **PFR** for all test programs across all applicable rules. At the bottom row, the average of each column is shown. Differences between the averages are statistically significant: using paired t-Test for means, **PFR**s for **RSR1-C** and **RSR2-C** are significantly different with $P(T \leqslant t) = 0.03$.

7. Analysis and discussion

7.1. Test program classification

Analytically, a test problem can fall into the following classes. Knowing how many are in each class can help to understand the value of the proposed approach.

Table 9
Test programs with missed paths.

Program	RSR1-B	RSR1-C	RSR2-B	RSR2-C
mmA2008		x		x
iA2008				
bisA2008				
mtA2008				
tA2008				
binA2008				
bubA2008				
gA2008		x		x
rA2008	x	x		
tM2004		x		x
eiR1985		x		x
qG1997				
ttB2002	x		x	
eiB2002				
qB2002		x		
scB2002		x		x
fcB2002				
fB2002	x	x		x
bG2011				
fG2011	x	x	x	x
sG2011		x		x

Table 10
Statistics of rules on **MP**.

Rule	MTP	MP	STD	Min	Max
RSR1-B	2	0.48	2.02	0.00	9.30
RSR1-C	0	0.84	1.69	0.00	7.08
RSR2-B	9	0.18	0.77	0.00	3.52
RSR2-C	2	0.17	0.42	0.00	1.81

Table 11
Statistics of rules on efficiency.

Rule	MTP	Eff. %	STD	Min	Max
RSR1-B	2	58.85	48.73	−90.80	96.18
RSR1-C	0	73.99	14.46	32.84	93.81
RSR2-B	9	8.74	78.28	−170.74	90.30
RSR2-C	2	31.96	23.03	−12.83	70.47

F1 All paths are feasible, and all are found before **GR** says to stop. For these programs, searching stops early anyway, so the proposed approach is just overhead. We have seen that the overhead is trivial, so this is not a problem.

F2 All paths are feasible, but **GR** would suggest stopping before some feasible paths are found. For these, the **GR** approach means there is a loss of performance, i.e. the assumption is wrong that all paths not found yet when **GR** says to stop are infeasible. Execution time is shorter, but something is lost, so this is a trade off.

I1 Some paths are infeasible, but all that are feasible are found by the time **GR** says to stop. The assumption that missed paths are infeasible is correct. **GR** saves time, and there is no loss of accuracy, so the proposed approach is beneficial.

I2 Some paths are infeasible, and some that are actually feasible get missed if searching stops at the time suggested by **GR**. The assumption that all remaining uncovered paths are infeasible is wrong. Like Class **F2**, it is a trade off of execution time for path coverage.

Table 13 presents the classification of test programs.

Two programs are in F1 with all combinations of parameters; the proposed approach neither helps nor hurts. Seven programs are in I1 with all combinations of parameters; the proposed approach is beneficial. One program (fG2011) is in F2 with all combinations of parameters; although all its paths are feasible, there are many of them and its search space is large. This is a hard problem, and feasible paths are also missed if the decision to stop is based on searching until **Gens** generations. There is no program that is in I2 with all combinations of parameters.

There are four programs that are in F1 with some parameter settings but in F2 with others; that is, with some parameter settings, the proposed approach causes some paths to be missed that could be found if searching continued for longer. For each of these programs, stopping after **Gens** generations also missed some feasible paths with some parameter settings.

In two programs with infeasible paths, using the weaker decision rule (RSR-1 instead of RSR-2) meant that some feasible paths were missed that were found when the stronger decision rule meant that searching continued for longer.

Table 12
Summary of PF and PFR.

Program	Paths		PF		PFR			
	All	Feas	Best	Comm	RSR1-B	RSR1-C	RSR2-B	RSR2-C
mmA2008	13	13	13.00	11.63	13.00	10.50	13.00	11.57
iA2008	6	5	5.00	5.00	5.00	5.00	5.00	5.00
bisA2008	9	9	6.00	6.00	6.00	6.00	6.00	6.00
mtA2008	52	20	19.00	17.53	18.88	17.54	18.87	17.53
tA2008	4	4	4.00	3.83	4.00	3.83	4.00	3.83
binA2008	7	7	7.00	7.00	7.00	7.00	7.00	7.00
bubA2008	15	4	4.00	4.00	4.00	4.00	4.00	4.00
gA2008	8	5	5.00	3.57	5.00	1.00	5.00	3.18
rA2008	5	4	4.00	3.97	3.97	3.73	4.00	3.97
tM2004	8	7	6.23	4.20	6.21	3.90	6.23	4.10
eiR1985	12	3	2.97	2.97	2.97	1.00	2.97	2.40
qG1997	21	4	4.00	4.00	4.00	4.00	4.00	4.00
ttB2002	8	8	8.00	7.03	7.53	7.03	7.87	7.03
eiB2002	31	5	5.00	4.90	5.00	4.90	5.00	4.90
qB2002	27	10	10.00	8.63	10.00	8.00	10.00	8.63
scB2002	15	4	4.00	3.83	4.00	1.00	4.00	3.82
fcB2002	5	5	5.00	4.90	5.00	4.90	5.00	4.90
fB2002	32	8	7.87	7.87	7.67	7.67	7.86	7.86
bG2011	20	11	11.00	11.00	11.00	11.00	11.00	11.00
fG2011	30	30	25.13	9.30	15.83	7.69	21.61	8.76
sG2011	32	32	32.00	17.77	32.00	10.69	32.00	15.96
Average	17.14	9.43	8.96	7.09	8.48	6.22	8.78	6.91

Table 13
Test programs classification.

Program	RSR-1B	RSR1-C	RSR2-B	RSR2-C
mmA2008	F1	F2	F1	F2
iA2008	I1	I1	I1	I1
bisA2008	I1	I1	I1	I1
mtA2008	I1	I1	I1	I1
tA2008	F1	F1	F1	F1
binA2008	F1	F1	F1	F1
bubA2008	I1	I1	I1	I1
gA2008	I1	I2	I1	I2
rA2008	I2	I2	I1	I1
tM2004	I1	I2	I1	I2
eiR1985	I1	I2	I1	I2
qG1997	I1	I1	I1	I1
ttB2002	F2	F1	F2	F1
eiB2002	I1	I1	I1	I1
qB2002	I1	I2	I1	I1
scB2002	I1	I2	I1	I2
fcB2002	I1	I1	I1	I2
fB2002	F2	F2	F1	F2
bG2011	I1	I1	I1	I1
fG2011	F2	F2	F2	F2
sG2011	F1	F2	F1	F2

In the remaining five programs, using common parameter settings meant that some paths were missed that would have been found if the best setting for each separate program had been used. For each of these programs, stopping after **Gens** generations also missed some feasible paths. That some paths were missed is not due to the proposed approach, but with the use of a general parameter setting. However, a common parameter setup makes most sense in practice. Finding the optimal parameter settings for a given problem makes little sense: by the time experimentation has found the best settings, the target paths that can be covered probably already have been covered, and the overall effort to search for the best parameter settings and then to search for test data with those settings may approach the effort of manual test data generation.

In summary, the proposed approach is irrelevant for two test programs, and beneficial for nine test programs (if the stronger decision rule is used). For nine test programs, whether or not feasible paths were missed depends on parameter settings, and the same is true of searching for a fixed number of generations. For one test program, feasible paths were missed regardless of parameter settings, and the same is true of searching for a fixed number of generations.

Detailed inspection of the results from each of the 30 runs with a given test program and parameter setup showed that if paths were missed, it was always the same ones. So, the difficulty of finding test data to cover a path does not change regardless of the treatments.

7.2. Decision rules

The combined rule RSR considers both the likelihood of covering further paths, and the stability of the predicted number of generations at which that likelihood reaches a desired level.

As $\lambda \to 0$ it is still possible that further paths might be found, but of course with very low probability. As long as $\Delta\theta$ is high, the predicted λ values over generations may be low but they are not stable. In other words, SR affects the rate of change of λ over generations.

As for the thresholds, smaller values mean less chance of missing feasible paths, but they also mean searching continues for longer, costing more computation time. On average over all test programs, the lower the value of λ the longer time required. Selecting the threshold should be based on the user's preference. So, it depends on how much the user wants to spend resources and is able to tolerate the chance of missing feasible paths.

RSR2-C appears to be the best decision rule in practice. The use of common parameters is more practical than finding the best parameters for each separate test program, and the low thresholds in the decision rule mean that few feasible paths are missed.

7.3. Efficacy and efficiency

Compared to searching for 100 generations, the average number of paths missed by **RSR2-C** is at most 0.6, and averages 0.1 (see Table 10). Computation time is reduced by an average of 30% (see Table 11). The proposed approach seems to be beneficial.

7.4. The effect of varying *Gens*

This analysis has compared performance with the proposed stopping condition against stopping at **Gens** generations, where **Gens** is set at 100. We have seen that in most cases, the stopping condition suggests stopping before 100 generations, and that few paths are missed.

Consider RSR2-C (common parameter setting, most stringent rule: the most straightforward to use in practice). Increasing **Gens** from 100 to 500 multiplies the total searching effort across the 21 test programs by 4.4 (less than 5, because some searches can stop early with all target paths found). Using the proposed stopping rule instead would now mean that 91% of the total searching time would be avoided, compared to searching for 500 generations. In 8 of the 21 test programs for which target paths are sometimes missed when **Gens** is 100, two still miss some paths sometimes when **Gens** is 500. The mean number of missed feasible paths, compared to searching for **Gens** generations, falls slightly from 0.1 to 0.07.

In general, as **Gens** increases, the efficiency of using the proposed stopping condition instead also increases, but it may be off-set by the possibility that more paths will be covered during the extra searching time. The example of increasing **Gens** from 100 to 500 shows that a substantial increase in average search time brings an increase in the average number of paths covered, but a small one. Increasing **Gens** further will continue to increase both the cost (execution time) and return (average number of paths covered), but we expect the returns to diminish.

7.5. Practical application

The evaluation reported above is based on expected performance over a representative number of runs, in order to understand the strengths and weaknesses of the approach compared to searching for **Gens** generations.

In practice, a tester is interested in obtaining a complete set of test cases. Average performance over 30 searches is not relevant, if at any point during those searches the full set of target paths has been covered during the aggregation of all runs so far. This cannot occur if any paths are infeasible, as in 14 of the 21 programs studied here.

The question then is how to recognize when to stop searching, judging with some level of confidence that as-yet-uncovered paths are infeasible rather than just hard to find.

One approach is to adjust the parameters that govern the search. The tester can decide how much searching time they are willing to expend in the search to cover target paths. If searching is to continue for a fixed number of generations, **Gens** can be set to as large a value as the tester can afford. If using our proposed stopping condition, $\Delta\theta$ and λ can be set to small values.

With either approach, it would be helpful to apply knowledge about what makes some target paths hard to cover [19] – for example, test cases in which all input values are the same – to create some specific test cases in the initial population (the rest of the initial population would still be generated randomly). This could make it more likely that paths that are otherwise hard to cover will be covered quickly, thus making it more likely that paths which remain uncovered after several generations are infeasible.

In practice, to use this proposed approach one would do exactly the same as if searching was based on **Gens**, in terms of defining target paths; setting GA parameters such as population size and mutation rate; stopping the search immediately if all target paths are covered; and deciding how many searching runs to do if some target paths remain uncovered. Instead of searching for a fixed number of generations, with **Gens** set to a large value, the proposed approach can be used with $\Delta\theta$ and λ set to small values. The advantage of our approach is that it provides a specific probability that searching for another generation will cover a new path.

8. Threats to validity

The following are considered to be challenges to make the approach widely applicable. Firstly, fine tuning the rule thresholds could be better with more tuning programs, because the results could be more generic and representative. However, the tuning programs used here represent a range of relevant program characteristics. So, we believe the thresholds used here are reasonable.

Secondly, testing with further programs that have different characteristics is needed. This is in order to gain more confidence that the approach will be successful for a range of programs. Again, the test programs used here display a range of characteristics; however, broader testing is a matter for further work.

The programs that have been studied here are all quite small (mean size is 33 LOC, maximum is 95 LOC; mean cyclomatic complexity is 7, maximum is 16). However, path testing applies at the level of logical paths through a single code module (method, function, or procedure), and that is what we study here. Code modules are usually smaller than 95 LOC, and usually have cyclomatic complexity smaller than 10, so we believe that the test programs studied here are representative in those respects.

We have only used test programs written in C. The language itself should not directly affect the applicability of the approach, since target paths are based on the control flow graph, which is independent of the language. There could be an indirect effect: if the type of problem being solved (which may depend on the application domain) makes a certain language particularly suitable, and the type of problem also affects the difficulty of finding suitable test data, the suitability of the method may vary between languages. Studying programs written in other languages is a matter for further work.

Many of the test programs studied in this paper are standard test programs for research in this area. Some test problems are taken from parts of real world problems, i.e. fG2011 and sG2011, but not many. The set of test problems should be broadened to include more real world programs, from various domains.

9. Conclusions

The proposed approach, inspired by software reliability growth models, is a promising approach to be used as a stopping criterion in evolutionary path testing. The justification and the experimental results have shown its feasibility.

In deciding the threshold values for rules, the user's preferences and the resources available should be considered. The over-riding objective is path coverage. As long as this is achieved, efficiency can be a consideration. The reliability model parameter values identified in this work, i.e. $\lambda \leqslant 0.1$ and $\Delta\theta \leqslant 0.0001$ (**RSR2-C**), are effective, with 32% less computation required on average over 21 test programs, at the average cost of missing 0.1 feasible paths, compared to searching for 100 generations.

With the computation power of modern computers, the saving in computation time may not seem much of an issue. The issue remains of how to decide that the value of further searching is too low (most likely because remaining uncovered paths are infeasible), even though the time to search for another generation may be low.

The main benefit of the proposed approach is that an arbitrary parameter (how many generations to search for) is replaced by a method that is based on the history of the search itself, and the tester's decision about when to stop testing can be based instead on the probability that further testing will cover no further paths, and the stability of that probability. There are still parameters to set (threshold values for λ and $\Delta\theta$ in the decision rules), but they are based on things that are directly meaningful to the tester.

Future work has several directions. The first is to broaden the generality of the results, by investigating programs with a wider range of internal characteristics in terms of logical structure and selection statement complexity. Others are to investigate the effect of using different forms of software reliability growth models, and to investigate the effect of varying other search parameters, particularly population size. Further, we intend to study programs written in other languages, and programs taken from industry.

Acknowledgments

Acknowledgments are due to the Indonesia General Directorate of Higher Education (IGDHE) of the Ministry of National Education of the Republic of Indonesia for providing a Ph.D. Scholarship, the University of New South Wales Canberra (UNSW Canberra) for providing a Completion Scholarship, and the Research Student Unit of the UNSW Canberra for supporting a Research Publication Fellowship.

References

[1] J.A. Whittaker, What is software testing? and why is it so hard?, IEEE Softw 17 (2000) 70–79. doi:http://doi.ieeecomputersociety.org/10.1109/52.819971.
[2] N. Mansour, M. Salame, Data generation for path testing, Softw. Qual. Control 12 (2) (2004) 121–136. doi:http://dx.doi.org/10.1023/B:SQJO.0000024059.72478.4e.
[3] P. McMinn, Search-based software test data generation: a survey, Softw. Test., Verif. Reliab. 14 (2004) 105–156.
[4] P.M.S. Bueno, M. Jino, Identification of potentially infeasible program paths by monitoring the search for test data, in: Proceedings of the 15th IEEE International Conference on Automated Software Engineering 2000 (ASE 2000), IEEE Computer Society, Grenoble, France, 2000, pp. 209–218. doi:http://dx.doi.org/10.1109/ASE.2000.873665.
[5] G.J. Myers, The Art of Software Testing, World Association Inc., 2004.
[6] M. Harman, S.A. Mansouri, Y. Zhang, Search Based Software Engineering: A Comprehensive Analysis and Review of Trends Techniques and Applications, Technical Report TR-09-03, April 2009.
[7] M. Harman, P. McMinn, A theoretical and empirical study of search-based testing: local, global, and hybrid search, IEEE Trans. Softw. Eng. 36 (2) (2010) 226–247, http://dx.doi.org/10.1109/TSE.2009.71.
[8] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, first ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.
[9] M. Pei, E.D. Goodman, Z. Gao, K. Zhong, Automated Software Test Data Generation Using a Genetic Algorithm, Tech. rep., Michigan State University, June 1994.
[10] J.-C. Lin, P.-L. Yeh, Using genetic algorithms for test case generation in path testing, in: Proceedings of the 9th Asian Test Symposium 2000 (ATS '00), 2000, pp. 241–246, doi:http://dx.doi.org/10.1109/ATS.2000.893632.
[11] J.-C. Lin, P.-L. Yeh, Automatic test data generation for path testing using gas, Inform. Sci. 131 (1–4) (2001) 47–64. doi:http://dx.doi.org/10.1016/S0020-0255(00)00093-1.
[12] I. Hermadi, M.A. Ahmed, Genetic Algorithm based test data generator, in: Proceedings of the 2003 Congress on Evolutionary Computation (CEC), vol. 1, 2003, pp. 85–91.

[13] M.R. Girgis, Automatic test data generation for data flow testing using a genetic algorithm, J. Univ. Comput. Sci. 11 (6) (2005) 898–915.
[14] J. Miller, M. Reformat, H. Zhang, Automatic test data generation using genetic algorithm and program dependence graphs, Inform. Softw. Technol. 48 (7) (2006) 586–605. doi:http://dx.doi.org/10.1016/j.infsof.2005.06.006.
[15] M.A. Ahmed, I. Hermadi, GA-based multiple paths test data generator, Comput. Oper. Res. 35 (2008) 3107–3124.
[16] Y. Chen, Y. Zhong, Automatic path-oriented test data generation using a multi-population genetic algorithm, in: Proceedings of the 4th International Conference on Natural Computation, 2008 (ICNC '08), vol. 1, 2008, pp. 566–570, doi:http://dx.doi.org/10.1109/ICNC.2008.388.
[17] Y. Chen, Y. Zhong, T. Shi, J. Liu, Comparison of two fitness functions for ga-based path-oriented test data generation, in: Fifth International Conference on Natural Computation, 2009, ICNC '09, vol. 4, 2009, pp. 177–181, doi:http://dx.doi.org/10.1109/ICNC.2009.235.
[18] Y. Chen, Y. Zhong, Experimental study on ga-based path-oriented test data generation using branch distance function, in: Third International Symposium on Intelligent Information Technology Application, 2009, IITA 2009, vol. 1, 2009, pp. 216–219, doi:http://dx.doi.org/10.1109/IITA.2009.232.
[19] I. Hermadi, C. Lokan, R. Sarker, Genetic algorithm based path testing: Challenges and key parameters, World Cong. Softw. Eng. 2 (2010) 241–244. doi:http://doi.ieeecomputersociety.org/10.1109/WCSE.2010.82.
[20] P.M.S. Bueno, M. Jino, Automatic test data generation for program paths using genetic algorithms, Int. J. Softw. Eng. Knowl. Eng. (IJSEKE) 12 (6) (2002) 691–709.
[21] J.D. Musa, K. Okumoto, A logarithmic poisson execution time model for software reliability measurement, in: Proceedings of the 7th International Conference on Software Engineering, ICSE '84, IEEE Press, Piscataway, NJ, USA, 1984, pp. 230–238. <http://portal.acm.org/citation.cfm?id=800054.801975>.
[22] B.F. Jones, H.-H. Sthamer, D. Eyres, Automatic structural testing using genetic algorithms, Softw. Eng. 11 (5) (1996) 299–306.
[23] P.M.S. Bueno, M. Jino, Automatic test data generation for program paths using genetic algorithms, in: Proceedings of the 13th International Conference on Software Engineering & Knowledge Engineering (SEKE '01), Buenos Aires, Argentina, 2001, pp. 2–9.
[24] P. McMinn, M. Harman, D. Binkley, P. Tonella, The species per path approach to search-based software test data generation, in: International Symposium on Software Testing and Analysis (ISSTA 2006), ACM, 2006, pp. 13–24, http://dx.doi.org/10.1145/1146238.1146241.
[25] M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, J. Wegener, The impact of input domain reduction on search-based test data generation, in: ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ACM, New York, NY, USA, 2007, pp. 155–164.
[26] K. Lakhotia, M. Harman, H. Gross, Austin: a tool for search based software testing for the c language and its evaluation on deployed automotive systems, 2nd International Symposium on Search Based Software Engineering 0 (2010) 101–110. doi:http://doi.ieeecomputersociety.org/10.1109/SSBSE.2010.21.
[27] A. Arcuri, A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage, IEEE Trans. Softw. Eng. 38 (3) (2010) 497–519.
[28] G. Fraser, A. Arcuri, EvoSuite at the SBST 2013 tool competition, in: SBST Workshop, 2013.
[29] I. Hermadi, Path Testing Using Genetic Algorithm, Ph.D. Thesis, University of New South Wales, Canberra, Australia, August 2012 (submitted for examination).
[30] D. Gong, W. Zhang, X. Yao, Evolutionary generation of test data for many paths coverage based on grouping, Syst. Softw. 84 (12) (2011) 2222–2233, http://dx.doi.org/10.1016/j.jss.2011.06.028. http://dx.doi.org/10.1016/j.jss.2011.06.028.
[31] T. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.
[32] G.J. Myers, An extension to the cyclomatic measure of program complexity, SIGPLAN Not. 12 (10) (1977) 61–64. doi:http://doi.acm.org/10.1145/954627.954633.
[33] E. Alba, F. Chicano, Observations in using parallel and sequential evolutionary algorithms for automatic software testing, Comput. Operat. Res. 35 (2007) 3161–3183.
[34] A.J. Offutt, J. Pan, T. Zhang, K. Tewary, Experiments with Data Flow and Mutation Testing, Technical Report ISSE-TR-94-105, ISSE, 1994.
[35] R.P. Pargas, M.J. Harrold, R.R. Peck, Test-data generation using genetic algorithms, Softw. Test., Verif. Reliab. 9 (4) (1999) 263–282.
[36] R. Blanco, J. Tuya, B. Adenso-Diaz, Automated test data generation using a scatter search approach, Inform. Softw. Technol. 51 (4) (2009) 708–720.
[37] C. Ramamoorthy, S.-B.F. Ho, W. Chen, On the automated generation of program test data, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 293–300.
[38] C.C. Michael, G.E. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Trans. Softw. Eng. 27 (12) (2001) 1085–1110. doi:http://dx.doi.org/10.1109/32.988709.
[39] J. Wegener, A. Baresel, H. Sthamer, Suitability of evolutionary algorithms for evolutionary testing, in: Proceedings of the 26th Annual International Computer Software and Applications Conference 2002 (COMPSAC 2002), 2002, pp. 287–289, doi:http://dx.doi.org/10.1109/CMPSAC.2002.1044566.
[40] J. Wegener, K. Buhr, H. Pohlheim, Automatic test data generation for structural testing of embedded software systems by evolutionary testing, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '02, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002, pp. 1233–1240. <http://dl.acm.org/citation.cfm?id=646205.682781>.
[41] R. Sagarna, X. Yao, Handling constraints for search based software test data generation, in: ICSTW '08: Proceedings of the IEEE International Conference on Software Testing Verification and Validation 2008, 2008, pp. 232–240.
[42] S. Rapps, E.J. Weyuker, Selecting software test data using data flow information, IEEE Trans. Softw. Eng. 11 (1985) 367–375.
[43] M.J. Gallagher, V. Narasimhan, Adtest: a test data generation suite for ada software systems, IEEE Trans. Softw. Eng. 23 (1997) 473–484. http://doi.ieeecomputersociety.org/10.1109/32.624304.
[44] N. Gupta, A.P. Mathur, M.L. Soffa, Automated test data generation using an iterative relaxation method, SIGSOFT Softw. Eng. Notes 23 (6) (1998) 231–244. http://doi.acm.org/10.1145/291252.288321.
[45] S. Artifact Infrastructure Repository (SIR), A Repository of Software-Related Artifacts Meant to Support Rigorous Controlled Experimentation, 2011 <http://sir.unl.edu/portal/index.html>.
[46] B. Korel, Automated software test data generation, IEEE Trans. Softw. Eng. 16 (8) (1990) 870–879.
[47] I. Hermadi, Genetic Algorithm based Test Data Generator, Master's thesis, King Fahd University of Petroleum & Minerals (KFUPM), June 2004.