Diversity oriented test data generation using metaheuristic search techniques

Paulo M.S. Bueno a,*, Mario Jino b,1, W. Eric Wong c,2

Abstract

We present a new test data generation technique which uses the concept of diversity of test sets as a basis for the diversity oriented test data generation – DOTG. Using DOTG we translate into an automatic test data generation technique the intuitive belief that increasing the variety, or diversity, of the test data used to test a program can lead to an improvement on the completeness, or quality, of the testing performed. We define the input domain perspective for diversity (DOTG-ID), which considers the distances among the test data in the program input domain to compute a diversity value for test sets. We describe metaheuristics which can be used to automate the generation of test sets for the DOTG-ID testing technique: simulated annealing; a genetic algorithm; and a proposed metaheuristic named simulated repulsion. The effectiveness of DOTG-ID was evaluated by using a Monte Carlo simulation, and also by applying the technique to test simple programs and measuring the data-flow coverage and mutation scores achieved. The standard random testing technique was used as a baseline for these evaluations. Results provide an understanding of the potential gains in terms of testing effectiveness of DOTG-ID over random testing and also reveal testing factors which can make DOTG-ID less effective.

Article history:
Available online 23 January 2011

Keywords:
Software testing
Test data generation
Random testing
Simulated annealing
Genetic algorithms
Simulated repulsion

1. Introduction

Software testing aims at revealing programs’ faults by executing the software with selected inputs and evaluating the correctness of the program behavior and output against the expected ones [56]. Because it is in general infeasible to execute all possible combinations of input values, the tester faces the problem of selecting a sub-set of input data from the program input domain to be used in testing.

Various techniques have been proposed for test data selection. Random testing (RT) randomly samples input values from the program input domain using a uniform distribution over the entire input domain, or a distribution based on the expected use profile [33]. Functional techniques use information about the program functionality to define classes of behaviors to be tested [7]; structural techniques consider the source code and establish structural components to be tested [46,56,57,65].

Part of this work was developed while Paulo Bueno was a visiting scholar at the University of Texas at Dallas working in Professor W. Eric Wong’s research group.

* Corresponding author. Tel.: +55 19 37466171.
E-mail addresses: paulo.bueno@cti.gov.br (P.M.S. Bueno), jino@dca.fee.unicamp.br (M. Jino), ewong@utdallas.edu (W.E. Wong).

1 Tel.: +55 19 35213783.
2 Tel.: +1 972 883 6619.
Fault based techniques suggest the selection of test data to uncover classes of faults commonly found in programs [24]. All these techniques are referred to as partition testing techniques because they explicitly divide the program input domain into partitions to be covered by at least one test datum [75]. Independently of the technique employed to generate the test data, it is necessary to determine whether or not the results of a test execution are correct. This evaluation requires a mechanism, referred to as an “oracle,” to decide whether the obtained output is the correct output [74].

To make the testing techniques applicable, effort has been expended to develop techniques for the automatic generation of test data which meet the requirements of the testing techniques. The first studies in test automation applied symbolic execution to compute and solve the “path constraints” in structural testing [19,38,64]. The dynamic approach was proposed as an alternative to overcome difficulties of symbolic computations of loops, arrays and pointers. In this approach, input data are assigned to input variables and the program execution is monitored; direct search methods are used to look for a set of values which cause the desired goal (or path) execution [29,42,54]. Model-based testing approaches use models that describe the expected software behavior, such as finite state machines, to be used for the automatic generation of test data and for evaluating whether the actual software behavior corresponds to that expected based on the model [20].

Recently, search and optimization techniques have been used as a tool for the automatic test data generation for several testing techniques. Genetic algorithms [1,30,53,44,69] are used for test data generation for statement, branch, and path testing and also for fault based testing [9,39,51,61]. Simulated annealing [26,41] is used for testing exception conditions [70]. Lately, particle swarm optimization [40] and ant colony [36] were also applied for (respectively) structural testing [77] and mutation testing [5]. The general recipe for these approaches is to design a measure that characterizes the goodness of a given solution to the test data generation goal and apply the metaheuristics to search in the program input domain for a test datum which achieves the desired goal.

This approach of search based software testing [50] is part of a broader effort of applying search techniques to solve software engineering problems: the approach “search based software engineering” (SBSE) [18,34,35] explores the potentially useful idea of bringing computational intelligence techniques (CI) to solve real world software engineering problems [62].

In this paper we present a new test data generation technique which uses a concept named “diversity of test sets” as a basis for the diversity oriented test data generation (DOTG) [10,11]. Diversity is a measure we propose which reflects the degree of variability among the test data in the set. Diversity oriented test sets (DOTS) are generated by an iterative optimization process applied on random test sets (RTS). We present the application of two known metaheuristics for generating DOTS – simulated annealing (SA) and genetic algorithms (GA) – and detail a new metaheuristic, simulated repulsion (SR), designed for this purpose. SR creates DOTS by simulating particle systems subject to electrical repulsion forces.

DOTG was proposed based on the intuition that a good test set should exercise the program under test as completely as possible and that variability, or diversity, among the test data in the set plays an important role in this completeness. This idea, embodied in partition and model based testing techniques, is not novel; rather, it is a well accepted guideline for test case design. Our contribution was to translate the intuitive idea into automatic testing techniques by the definition of diversity measures and suitable algorithms to generate test sets.

The diversity concept can be defined under different perspectives, each one considering a distinct type of information to determine diversity values for test sets and to be used for generating test sets with high diversity according to the desired perspective. For instance, a structural perspective (SP) could measure the diversity of the allocation of test data in exercising the internal control and data flow structures of the program under test. By using such a measure, a suitable optimization algorithm could automatically generate a test set which allocates test data in a balanced way to cover each def-use association [65], possibly using several different input values for covering each association and also forcing the coverage of associations in different execution sequences when the program allows such executions. This SP can also be designed to consider information from fault-based techniques, such as mutants killed in mutation testing [24] to compute diversity values for the test sets. In such a scenario, DOTG can serve as an integration framework for the automatic generation of test sets according to different testing techniques.

In this work, however, we restrict ourselves to the definition and analysis of the input domain perspective for diversity (here named simply DOTG-ID), which considers the distances among the test data in the program input domain to compute a diversity value for test sets. DOTG-ID does not require elaborate information about the program under test or about the coverage attained in the execution of the test cases. Nevertheless, as our empirical evaluation indicates, even without using such sophisticated, and expensive, information the input domain perspective for diversity can lead to potential gains over the standard random testing.

We performed initial evaluations of the DOTG-ID. In the first evaluation we performed a Monte Carlo simulation to assess how test set size, failure rate, and type of failure pattern affect the effectiveness of DOTG. In the second evaluation, using the data flow based testing tool Poke-Tool [12] and the mutation based testing tool Proteum [23] we determined the coverage and mutation scores achieved by RTS and DOTS for three programs. Both the simulation and the coverage results provide an understanding of the potential gains of DOTG-ID over the standard RT and also reveal situations in which RT is a better choice.

The rest of the paper is organized as follows. In Section 2 we describe the DOTG-ID testing technique and the underlying concepts of the technique. Section 3 discusses related works. Section 4 describes the algorithms for generating DOTS, with emphasis on the new SR heuristic. Section 5 describes results from empirical evaluations carried out to assess the technique effectiveness. Section 6 concludes the paper.
2. Diversity oriented test data generation

The initial conjecture for developing the DOTG-ID was that sampling the program input domain in a systematic way, increasing the diversity of values for the input variables and, consequently, the uniformity of the coverage of the input domain, could improve the test quality over the standard random sampling. This conjecture was first inspired by the fact that increasing diversity, or uniformity of sampling, leads to an improvement in performance in other fields: quasi-random integration, a Monte Carlo integration, uses quasi-random numbers, instead of random, to increase the accuracy of the evaluation of integrals [55]; sampling based motion planning, which searches for collision – free paths connecting initial and final points in a space with obstacles, applies low dispersion sampling on the space to decrease the cost of planning [45]; the human immune system improves its pathogen recognition ability by using diversity generation mechanisms to optimize the coverage of the space of possible pathogens [21,60]. Surely all these sampling problems have some particular characteristics which are different from the ones in software testing; however, we believe that similarity is enough to justify the investigation of our initial conjecture. Following we discuss software testing concepts related to DOTG-ID.

2.1. Random testing

Random testing has been used by practitioners because it is a technique relatively easy to automate and to apply [33]. RT does not use elaborate information about the software, such as requirements, models, or implementation, to define testing requirements (or partitions) to be satisfied (or covered). This limited use of specific information about the software could lead to the conclusion that the technique’s effectiveness is not competitive when compared to more systematic techniques, such as functional, model based, or structural techniques. Empirical and analytical work has demonstrated that, on the contrary, the effectiveness of RT is quite similar to that of the systematic techniques and, given its lower cost, RT has potential as a useful technique [25,31,58,75].

2.2. Software failure

A failure region is a subset of the program input domain on which the software produces a faulty result [66,72]. To reveal a program fault, it is necessary to generate a test point placed in any failure region. Ammann and Knight, in their data diversity approach to fault tolerance, observe that a failure region is locally continuous [3]. Bishop remarks “strong experimental evidences on the existence of “blob defects””, defects which occupy contiguous regions of the input space. All 30 defects analyzed in his work present contiguous regions of failure [8]. Hamlet points out a similar effect called failure continuity: “if the program fails to meet its specification at some input, then it also fails in a neighborhood” [32].

Assuming this failure continuity holds to some degree in a program, that is, most of the failure regions have some extension, it makes sense to avoid using two test data too close to each other. In this case, if the software passes for the first test datum, implying that the test datum is out of a failure region, it will likely also pass for the second test datum, likely also out of the failure region. This second test datum could be better used in some place which is far from the first test datum.

The diversity measure we propose is a way to improve the effectiveness of RT by avoiding the piling up of test data in parts of the input domain and consequent “empties” of test data in other parts. Increasing the diversity of values for the input variables and, consequently, the uniformity of coverage of the input domain, is a useful way to sample the input domain for testing purposes.

2.3. Diversity measure

We refer to a program unit (function or procedure) as a program $P$. An input variable for $P$ may be an input parameter, a global variable used in $P$, or a variable that appears in an input statement $\text{read}(x)$. The domain $D_x$ of the input variable $x$ is the set of all the values $x$ can take. The input domain of $P$ is the cross product of the $n$ input variables’ domains $D = D_{x_1} \times D_{x_2} \times \ldots \times D_{x_n}$, where $n$ is the number of input variables. A test datum, or input datum (t), is a combination of values assigned to input variables $(t = x_1, x_2, \ldots, x_n)$. Each $x_i$ represents one input variable and $t$ represents a single point in the $n$-dimensional input domain $D$. A test data set (or test set) is a collection of test data ($T = t_1, t_2, \ldots, t_m$); each $t_i$ represents one test datum and $m$ is the test set size. Consider two test data $x$ and $y$: $x = x_1, x_2, \ldots, x_n; y = y_1, y_2, \ldots, y_n$. We define the distance between two test data as:

$$D(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \quad (1)$$

$D(x, y)$ represents the Euclidean distance between the test data $x$ and $y$ in the $n$-dimensional program input domain. Consider a test set ($T$) composed of $m$ test data ($t$): $T = t_1, t_2, \ldots, t_m$.

We define the diversity of $T$ as:

$$\text{Div}(T) = \sum_{i=1}^{m} \sum_{j=1}^{m} \min(D(t_i, t_j)). \quad (2)$$
where \( \min(D(t_i, t_j)) \) returns \( D(t_i, t_j) \) if \( t_j \) is the closest test data to \( t_i \) (i.e., if \( D(t_i, t_j) \) is the minimum value for all \( j \)), and 0, otherwise. Hence, the diversity value of a test set \( \text{Div}(T) \) is the sum of distances between each test datum in the set and its nearest neighbor.

Fig. 1 shows three different placements of five test data in a two-dimensional input domain \((a \times a)\). In case 1 two test data are superimposed at the lower right corner of the input domain; in case 2 these two test data are close to each other \((a/8\) apart) and in case 3 each test datum in the set clearly explores a different part of the input domain. The diversity values computed for these test sets are: \( 3a \) \((3.000)\) for case 1; \( 25a/8 \) \((3.125)\) for case 2 and \( 5a\sqrt{2}/2 \) \((3.536)\) for case 3, the maximum diversity of the three cases (numerical values for \( a = 1 \)).

Characteristics of high diversity test sets are: (i) test data are placed uniformly in \( D \) (distances between each test data and its neighbors in \( D \) have low discrepancy); and (ii) the extreme points in each dimension of \( D \) are covered by test data in the set; i.e., the test set covers the domain’s boundaries. Fig. 2 shows a random test set and a diversity oriented test set, both with 100 test data.

Consider a program \( P \) with input domain \( D \) and a test set \( T \) of size \( m \). It is illustrative that in the particular case where \( m = |D| \) (the number of test data is equal to the number of points in the input domain), the unique test set with maximum diversity places each test datum at a different point of \( D \), the obvious optimal allocation, easily identified by a human tester. If a fault is hidden behind a single input point, only the test set with maximal diversity guarantees it to be revealed, and decreasing the diversity of \( T \) implies lowering the chance of revealing the fault. The test allocation that places all test data in the same point, the worst allocation, has the minimal diversity, equal to zero, and its chance to find the fault is only \( 1/|D| \).

Clearly, for most, if not all, testing problems \( m \ll |D| \). In such real cases the technique systematically combines a range of values for the input variables to eliminate the large voids of non-tested portions of the input domain (such as in cases 1 and 2, Fig. 1). The extent of these ranges of values depends on the size of the input domain \((|D|)\) and the size of the test set \((m)\). The larger the input domain, the wider the ranges of values for a given test set size. Conversely, the larger the test set size, the narrower the ranges of values for a given input domain size.

It is important to note that other distance measures, other than Euclidian distance, could be employed to compute diversity. Manhattan, Hamming or edit distances could also be used to create DOTS. Moreover, diversity measures from other fields (e.g., biological diversity of species [47] also could be investigated to create such test sets. Our choice to use Euclidian measures is due to its simplicity and the fact that it is also used to measure sizes of input domains and failure regions.

![Fig. 1. Different placements of test data and diversity.](image1)

![Fig. 2. RTS (left) and DOTS (right), |T| = 100.](image2)
2.4. Semantic fault size, test granularity and diversity

The semantic size of a given program fault is defined as $|D_f|/|D|$ where $|D|$ is the cardinality of the entire input domain and $|D_f|$ is the cardinality of the subpart which results in a faulty output [6,59]. It represents the relative size for which the output mapping is incorrect. Because of the continuous nature of software failure regions (as described on Section 2.2) this subpart $D_f$ will likely form a continuous failure region in $D$.

Since diversity oriented test sets present close to uniform distances among the test points, they allow a more precise control over the test set granularity. Variations in the test set size directly reflect the test set granularity and, therefore, the expected size of failure regions to be revealed by the test set. As the distances among neighboring test data are very similar, the value $Div(T)/|T|$, where $Div(T)$ is the diversity of test set $T$ and $|T|$ is its size, represents the size of the gap between adjacent test points in $D$. The failure regions in these gaps can hide from the test data, and therefore escape discovery. Increasing smaller gap sizes, obtained by increasing $|T|$, lead to the expectation of revealing progressively smaller failure regions. The test set forms a multi-dimensional sieve whose granularity, controlled by $|T|$, determines the expected size of the failure regions filtered in the program input domain. This “sieve behavior” is observed in the simulation (Section 5.1.1) in which the effectiveness reaches a maximum when the value of $Div(T)/|T|$ approaches the side length of block-shaped failure regions.

3. Related work

Approaches in the literature have aspects in common with the DOTG technique: antirandom testing [48,73], adaptive random testing [13,14,16,17], quasi-random testing [15] and lattice-based testing [49].

Antirandom testing generates test sequences (called antirandom test sequences) such that each new test datum $t_i$ (named test vector) in the sequence is chosen to maximize the total distance between $t_i$ and each previous test data $(t_0, t_1, \ldots, t_{i-1})$. Cartesian and Hamming distances are used in the attempt to find a test datum that is as different as possible from all previously applied test data. Exhaustive search is used for evaluating all possible remaining points in the domain for defining this datum. Properties of antirandom sequences are defined and used for reducing this cost [73].

Independently of the metaheuristic used to create DOTS, the DOTG-ID approach works with the entire test set. On the other hand, the antirandom technique performs a search for each test datum to be added in the sequence. Another important difference is related to the strategy of making each test datum as distant as possible from all the previously executed ones. This strategy (instead of the strategy of making a test datum as distant as possible only from the closest test datum) tends to push all the test data to the extreme points of the input domain. For example, consider the three cases of placements for the five test data in Fig. 1; the placement that maximizes such a measure used to generate antirandom sequences is the one of case 1 rather than the one of case 3.

Adaptive random testing (ART) modifies the standard random testing by using a procedure to select each new random test datum to be added to the test set. The technique defines two test sets: the executed set, containing the test data that have been executed, and the candidate set, containing a set of randomly selected test data. The executed set is then incrementally updated with the elements selected from the candidate set. In each increment the new test datum selected is the one that has the maximal value for the minimal distance among the distances to each test data in the executed set [16].

Several variations of the technique present different methods for selecting the next test datum to be added to the test set. For example, restricted random sets exclusion zones around all previous test cases and randomly generates new test cases out of the exclusion zones [13]. Mirror adaptive – the domain is partitioned and test cases generated (by using the standard ART) in one sub-domain are mirrored successively into other sub-domains [17]. Dynamic partitioning – uses the most recently performed test to subdivide the input domain, and selects test cases randomly from the largest remaining sub-domain [14]. Quasi-random testing uses quasi-random sequences, which are randomized (scrambling method) to allow the generation of many different sequences [15]. Lattice-based adaptive testing selects test cases based on a lattice structure in which test points are randomly translated by a small amount [49].

These testing techniques share with DOTG the intent of improving the test effectiveness by spreading the test data over the program input domain. However, we can recognize in DOTG some singular characteristics when compared to the described techniques.

In DOTG the random test sets are incrementally enhanced by repositioning the test data in the set – at each iteration of the test data generation algorithms (next section) – in a way that improves the diversity values of these sets. In our observations, these refining movements are essential to achieve a “global” high diversity of input values and to avoid parts of the domain with low diversity (pile up of test data). The techniques mentioned above do not allow repositioning: a bad choice for a test datum location cannot be fixed in a second step. Also, they generate a test datum using information on the positions of the previously generated test data, instead of the positions of test data of the whole test set, as performed in DOTG.

Our data and analysis suggest that DOTS provides effectiveness gains at a level comparable to other testing techniques such as antirandom testing, adaptive random testing, quasi-random testing and lattice-based testing. A more complete comparison among these techniques is currently underway. However, we would like to highlight the following important aspects that, regardless of any direct comparison, distinguish our technique from those mentioned above:

- The diversity-oriented approach can be defined by taking several kinds of information into account. This provides different perspectives for diversity. On the other hand, the other techniques only present a single perspective (how the test
points cover the program domain). Therefore, DOTG can be seen as a “general diversity approach” while ART, antirandom, quasi-random and DOTG-ID are “specific diversity approaches” focused on the program input domain.

- While other techniques are used for selection of test data, ours goes beyond as it also offers a way to evaluate the diversity of a test set. More precisely, our technique is unique in that it helps testers measure the diversity of a test set generated by using whatever other test selection criterion. E.g., if two different test sets satisfy the branch coverage criterion for a program, the diversity values of the test sets could be computed and be used as an orthogonal criterion to choose which test set to use. This characteristic broadens the circumstances in which our technique can be applied.

- Given the diversity value of a test set and its size, one can use our technique to find the average gap between two adjacent test data in a multidimensional program input domain. This information can then be used to predict the size of failure regions found by the test set (Section 5.1.1). Despite the fact that the practical use of this information still requires elaboration and further research, to our knowledge our technique is the first that establishes a relation between a measurable property of the test set (the diversity of the test set) and the size of failure regions to be found.

Since our first publication on the diversity oriented technique [10], proposals have been presented exploring ideas and techniques which are similar to ours [2,27].

Feldt et al. propose a model of test variability used to stress the selection of tests that are diverse from previous tests [27]. The proposed metrics calculates the distance between two test vectors using Information Distance, a metrics based on the Kolmogorov complexity. As such metrics is uncomputable, they propose using data compressors to obtain an approximation to Information Distance values. The authors claim that this metrics captures the “cognitive differences” among tests based on information of the tests traces. Such traces can represent different information, such as control flow, or data state changes. It is not clear, however, how the traces information is structured for computing the distances.

Application of the metrics using 25 test cases for the triangle classification program is presented. Results on how the test data are clustered by using the metrics are compared to how three human subjects evaluate and cluster the same test data. Conclusions point out that the metrics found clusters similar to those identified by humans.

No algorithms are proposed to be used for generating the test data using the proposed model. Also, there are no evaluations about the effectiveness of the generated test data. As the authors recognize, it is unclear whether differences on the information theoretical stand point are relevant for the test quality and whether a test that appears diverse to humans is more effective for testing purposes.

Alshraideh et al. present a heuristics for directing the search when the cost function for the test goal is unable to distinguish between candidate test inputs [2]. This problem may arise in search based test generation approaches that apply a cost function that evaluates tests with respect to achieving a given goal, e.g., traverse a program branch. If the branch predicate leads to a cost function that returns a constant value, it is difficult to discriminate between candidate inputs. The proposed heuristics is, in such a case, applied to search for inputs that produce scarce data states. The rationale is that commonly occurring data states will already have been encountered during the search, while scarce data states are more likely to produce new cost values.

Data scarcity search is initiated when the branch cost functions have become constant during the search. Values of variables (data states) produced during the executions are recorded, grouped into classes, and represented as sets of histograms. Such histograms are used to compute distance measures, which are applied in a cost function to lead to scarce data states. Their approach was evaluated empirically providing an indication that the approach is effective for guidance in the search in the set of studied programs.

A significant difference between our work and Alshraideh et al. heuristics is that data scarcity is based on the examination of intermediate program data states rather than on the input data. The goals of the approaches are also different; while we aim at increasing the effectiveness of testing in terms of fault revealing capability, their heuristics aims at giving guidance in the search for specific cases where a cost function returns constant values.

4. The automatic generation of diversity oriented test sets

The problem of generating diversity-oriented test sets in this input domain perspective is similar to the known NP-hard problems: sphere packing [68] and the spreading points [28], respectively – assemble the spheres in such a way that they do not overlap with each other and the sum of the volumes of the spheres packed is maximized; and arrange \( n \) points into a square so that the minimum distance among them is the greatest.

Therefore, we addressed the practical problem of generating DOTS by using metaheuristic techniques: a simulated annealing algorithm, an optimization method based on thermodynamic considerations which simulate the cooling of solids; a genetic algorithm, inspired by mechanisms of evolution and natural genetics; and the simulated repulsion algorithm (SR), proposed in this work, which is based on particle systems dynamics [78]. Following we briefly describe the concepts behind each metaheuristic and detail how each one is applied to automate DOTG-ID.

4.1. Simulated annealing

Simulated annealing is a method for solving difficult optimization problems [26,41]. It is based on an analogy with thermodynamics and the way that liquids freeze and crystallize. At high temperatures the molecules of a liquid move almost
freely. If the liquid is cooled slowly enough it gradually loses mobility and often forms a pure crystal that is completely or-
dered – the (global) minimum energy state for the system. The freezing process is guided by a cooling schedule that controls
the decay of the system temperature.

SA starts with an initial solution, often chosen at random. A neighbor of this solution is generated by a rearrangement
operation and the change in the objective function \( f \) is calculated. If an increase in \( f \) is found the current solution is replaced
by the neighbor (considering the goal of maximizing \( f \)); however, if \( f \) decreases, a probabilistic decision for the acceptance of
the worst solution (a downhill move) is determined by a sequence of random numbers. The probability of acceptance of a
change that causes a decrease of \( \delta \) in the function \( f \) is defined by the acceptance function:

\[
e^{-\frac{\Delta f}{T_p}}
\]

where \( T_p \) is the temperature control parameter. This implies that small decreases in \( f \) are more likely to be accepted than large decreases; and yet, when
\( T_p \) is high most moves to worse solutions (a smaller value for \( f \) will be accepted, but with \( T_p \) close to zero almost all moves to
worst solutions will be rejected. By sometimes allowing movements for worse solutions the SA algorithm attempts to avoid
becoming trapped in a local optimum.

4.1.1. Simulated annealing for test data generation

Our implementation is based on the algorithm simulated annealing for diversity (SAND), proposed as an immunological
approach to initialize neural network weights [22]. The SA algorithm starts with one randomly selected test set, composed of
a specific number of test data, and generates a new test set by making a random change in some test data. In our case this
change is performed by randomly selecting input variables from the test data with a mutation probability \( m_p \) and changing
their values by a randomly generated one, of the same type and inside the allowed range of values for the input variable.

The variation of diversity from the original to the changed test set is computed, applying Eq. (2) to each test set and com-
puting the difference between the values. If the difference is positive, i.e., a higher diversity, this changed test set becomes
the current solution for the next iteration of SA. Changed test sets, with lower diversity, can also be accepted as current solu-
tions. This acceptance is controlled by a function that uses random numbers and the temperature \( (tp) \) parameter to accept
the worse test sets with a controlled probability (Eq. (3)).

\[
\text{random}(0, 1) < e^{-\frac{\text{Div}(\text{New Test Set}) - \text{Div}(\text{Test Set})}{\text{C}_0}}
\]

A cooling schedule defines the reduction of \( tp \) values along the search and influences how test sets can be accepted at each
phase of the search. Our SA reduces the value of \( tp \) at each \( ks \) iteration by using \( tp = tp \cdot x \). Values for the initial temperature
(\( tini \)), for \( ks \) and for \( x \) were defined by running the algorithm and monitoring the performance in several different circum-
stances (typical values used: \( x = 0.95; \) \( ks = 100; \) \( tini = 1000 \)). The search runs until a lower bound for \( tp \) is reached, or it can be
stopped when no progress in \( \text{Div} \) values is observed for a given number of iterations.

The SA accepts as input the parameters: number of test data \( (NTD) \); number of input variables in each test data \( (NIV) \); the
description of each input variable of the test data: type, lower and upper limit values for the values; and search parameters
\( (tini, ks, mp) \). At the end of the search the DOTS is outputted to the tester.

4.2. Genetic algorithms

Evolutionary computation techniques (EC) apply the natural process of evolution as a problem-solving paradigm [52]. Ge-
genetic algorithms (GAs), part of the EC field, are search methods inspired by the mechanism of evolution and natural selection
[30,37,53,69]. GAs work with a population of potential solutions to a problem encoded as strings named individuals. Each
individual is formed by genes, each one assigned to a variable to be considered in the optimization problem. An evaluation
of the potential solutions is performed: a fitness function is used to quantify the quality of the solutions encoded in each
individual is formed by genes, each one assigned to a variable to be considered in the optimization problem. An evaluation
of the potential solutions is performed: a fitness function is used to quantify the quality of the solutions encoded in each
individual.

Individuals are selected for the next generation, based on the fitness values, and genetic operators are applied to create a new population of solutions to be used in the next iteration of the algorithm. The iterative selection and combi-
nation of individuals leads to progressively better individuals along the search.

4.2.1. Genetic algorithm for test data generation

The genetic algorithm works with an initial population of randomly generated test sets; each test set is composed of a
specific number of test data and each test datum is formed by the defined input variables. The algorithm iteratively selects
the test sets for the next generation by assigning a selection probability based on each test set diversity, computed by using
Eq. (2). The selected test sets undergo the crossover operation (exchange of values for test data between pairs of test sets)
and mutation (random changes of test input variable values). The iterative recombination and selection of test data moves
the search towards higher diversity test sets.

4.2.1.1. Selection, Crossover, and Mutation. In our GA we apply standard well-accepted genetic operators. At each generation
proportional selection chooses the better individuals for the next generation; uniform crossover is used for the exchange of
genetic information between individuals; and simple mutation introduces new values for selected genes (alleles).

The fitness value for each test set (individual) is the diversity of the test set, computed by using Eq. (2). The proportional
selection operator mimics nature’s “survival of the fittest” phenomenon. A test set \( T_i \) is selected for the next generation by
using a roulette wheel which makes the chance of the test set selection be proportional to the rate between its fitness
\( P(T_i) = \frac{D_{iv}(T_i)}{A_{vgDiv}} \)  

The uniform crossover is performed by randomly exchanging values of input variables, which form the test data whose collection is the test set, between a pair of test sets. The input variables values are exchanged with an independent probability \( pc \). The mutation operator is responsible for the introduction of new values for the input variables in the population of test sets, with probability \( pm \). It changes the original value of an input variable by a new randomly generated value inside the domain of the variable and according to its type. It is applied with an independent probability \( pm \) for each input variable in the test set.

The GA accepts the parameters: number of test sets (\( NTS \)); number of test data (\( NTD \)); number of input variables (\( NIV \)); description of each input variable of the test data: type, lower and upper limit values for the values; and search parameters (\( pm, pc \), and upper limit for the number of iterations). Values of input values are randomly generated for each input variable of each test datum and each test set. The randomly generated test sets are evaluated by computing each test set’s diversity values. Test sets are then iteratively selected (with a chance proportional to their diversity values) and modified with the application of genetic operators. At the end of the search the generated DOTs are outputted for the tester.

4.3. A specific self-organizing algorithm for improving diversity

SA and GA are powerful tools for solving complex generic problems; they can be applied independently of the function to be optimized, defined by the characteristics of the specific problem. However, it is known that if one captures specific knowledge about the problem, he/she can use this knowledge to enhance significantly the heuristic’s performance [71]. The SR algorithm captures the core behavior for generating diversity in the functional perspective of DOTG and presents a significant improvement on the effectiveness and efficiency over the SA and GA algorithms. Instead of using the diversity function (Eq. (2)) to drive the search process as done by SA and GA, SR works through an emergent behavior of self-organization of the test data in the program input domain, which is created by simulating particle systems [78].

4.3.1. The simulated repulsion algorithm

The SR algorithm is inspired by particle systems dynamics that are applied to problems from several contexts including nonlinear optimization problems, which are solved by simulating the social behaviors of swarms, such as bird flocks, fish schools or humans [36,40]. Instead of a social metaphor, we apply a physical metaphor by using particle systems. We simulate a physical system that, because of its characteristics, makes each test datum (a particle in the system) spontaneously find its place in the program input domain to maximize the diversity of the test set.

Each test datum in the set receives a positive electric charge (\( Q \)) whose initial value is a parameter of the SR. These charges cause the test datum to create an electrical field around itself. The electrical fields altogether result in forces applied on the test data that cause their movements inside the input domain.

The algorithm handles \( NTS \) test sets \( T(T_1, T_2, \ldots, T_{NTS}) \). Each test set contains \( NTD \) test data \( (t_1, t_2, \ldots, t_{NTD}) \). Each test data has \( NIV \) input variables \( x(x_1, x_2, \ldots, x_{NIV}) \). \( NTS, NTD, NIV \) and the description of each input variable \( x \) are also parameters of the SR algorithm. The description of each input variable includes its type (integer or float) and range of values that it can hold, e.g., from 0 to 5000. The ranges of values for the variables are assumed to be continuous.

Using a uniform random number generator, the algorithm generates \( NTS \) random test sets. Each test datum in each test set represents a particle placed in the \( NIV \)-dimensional program’s input domain \( (D) \). The value of each input variable of the test datum defines the test datum position with respect to one dimension of \( D \). Each test set \( T \) is a particle system that evolves independently from and concurrently with the other test sets. In each iteration of the SR algorithm, all \( NTS \) test sets have the positions of their test data updated according to the same physical rules. Forces between two test data are defined by an adaptation of Coulomb’s Law: the magnitude of the electrostatic force between two point electric charges is directly proportional to the product of the magnitudes of each charge and inversely proportional to the square of the distance between the charges:

\[
F(t_i, t_j) = \frac{Q^2}{d(t_i, t_j)^2},
\]

where \( Q \) is the current value of electric charge for the test data and \( d \) is the distance between \( t_i \) and \( t_j \). The resultant force \( RF \) on a given test datum \( t_i \) is the vector sum of the forces induced on \( t_i \) by all other \( t_j \) test data in the set \( (1 \leq j \leq NTD, i \neq j) \).

\[
RF(t_i) = \sum_{j=1}^{NTD} F(t_i, t_j).
\]

From one iteration to the next in the SR algorithm all test data of all test sets have their positions updated. The new position of a test datum is computed by using an adaptation of Newton’s mechanics:
\[ t_{\text{new}} = t_i + \frac{(RF(t_i))}{m}, \]  

where \( t_{\text{new}} \) is the new position of test data \( t_i \), \( RF(t_i) \) is the resultant force induced on \( t_i \) by the other test data in the set, and \( m \) is the constant mass of all test data. The mass of a test data (a small value, e.g., \( m = 10 \)) simply restricts the test data movements, that would tend to infinite with \( m = 0 \). If \( t_{\text{new}} \) is outside of \( D \) after the movement, an elastic collision is simulated. In such a case, the specific dimension, or dimensions, in which \( t_{\text{new}} \) would be placed outside \( D \) works as a barrier with which the test data collide and bounces back inside the input domain. The collision is dissipative in the sense that the new position of the test data in relation to the collided surface(s) after the collision is closer than before the movement. The elastic coefficient is a parameter set at around 0.9.

Snapshots of a test set at each iteration shows the test data closer to domain boundaries to be pushed towards the boundary, collide in the boundary and return back to a position closer to the boundary than before. Test data placed in the middle of the input domain are subject to lower resultant forces and move more slowly. As the search progresses, the resultant forces on each test datum decrease and the total amount of movement also decreases. In the final stages the systems of particles (test sets) reach a state close to the equilibrium of forces, the test data movements tend to stop, and the diversity value for the test set stabilizes.

Fig. 3 details the SR algorithm. Line 1 – SR accepts the parameters: number of test sets; number of test data; number of input variables. Line 3 – SR accepts the description of each input variable of the test data: type, lower and upper limit values for the values. Line 4 – SR accepts the initial value of charge for the test data; the upper limit for the number of iterations; and the discharge coefficient. In line 6, the value is defined for the mass of each test datum (can be any small value). Line 10 – values for input values are randomly generated for each variable of each test datum and each test set. The loop on line 12 controls the iterations of SR. In each iteration the distances of each test datum to all other test data are computed – lines 19 and 20. The total force applied on the test datum is computed – line 20, and is decomposed with respect to each dimension of \( D \) (related to each input variable) – line 22. The new position of the test datum with respect to each dimension is computed – line 25. The test datum is checked whether it would be outside of the input domain with respect to any dimension; in such a case, the specific dimension, or dimensions, in which \( t_{\text{new}} \) would be placed outside \( D \) works as a barrier with which the test data collide and bounces back inside the input domain. The collision is dissipative in the sense that the new position of the test data in relation to the collided surface(s) after the collision is closer than before the movement. The elastic coefficient is a parameter set at around 0.9.
case an elastic collision is simulated. The diversity of each test set is computed – line 31, the charge values for the test data are reduced – line 33, and the iteration counter incremented – line 34. In line 35 the new test sets are taken for the next iteration. At the end of the search DOTS are outputted to the tester.

It should be noted that all movement rules are, in fact, adaptations of physical laws: not expected to create realistic movements, but to result in the desired final placement of test points. The movement rules are based on Newton’s law: \( a = \left(1 \times m \times f \right) \); \( a: \) acceleration, \( f: \) force, \( m: \) mass. In SR the new positions of test data are computed based on acceleration values, instead of taking acceleration and speed into account. In collisions of test data at the boundaries of \( D \) the new positions of the test data are always closer to the boundary than they were before the collision. Collisions between test data are not considered.

The definition of initial values for the charge \((Q)\) requires some attention: a very small \( Q \) makes the movements too slow, while values too large for \( Q \) (higher than a \( Q_{\text{limit}}\)) seem to push the systems towards an unstable process in which the diversity values do not grow. We observed that for a given \( D \) the values for \( Q_{\text{limit}}\) are inversely proportional to the size of the test sets \(|T|\).

Our approach is to set \( Q \) by analogy taking \(|T|\) into account. Given that \( Q \) is below \( Q_{\text{limit}}\), which is easy to note by examining the diversity evolution along with the iterations, the search effectiveness is not very sensitive to variations of \( Q \). A discharge schedule decreases the value of the charge and the magnitude of forces along the search by an amount defined by the discharge coefficient (around 0.95); this fine-tunes the movements in the final stages of the search.

5. Empirical evaluation

Our primary concern in the evaluations is to assess the effectiveness of the algorithms for generating DOTS. Using SA, GA and SR we performed a number of test set generations to define the algorithm parameters that lead to a good performance. We then used the algorithms to evaluate their behavior with respect to some aspects, such as type of variables in the test data (integer, real and combinations); number of input variables in each test data (which characterize the dimension of the program input domain – the search space); and the size of test sets. The next step was to quantitatively evaluate the efficiency and effectiveness of the algorithms: we performed several test set generations and assessed the attained diversity values for defined number of iterations of the algorithms. This analysis allowed the conclusion that, in the current implementations, SR outperforms GA, which outperforms SA. As an instance, Table 1 shows the average of attained diversity values for a square input domain, with two real variables ranging from 0 to 1024 and test size \(|T|\) of 100. Diversity values were calculated by using Eq. (2). The diversity value for RTS is about 5100. As can be noted, SR achieves in 50 iterations diversity values greater than those achieved by SA and GA in 4000 iterations. In fact, with just 10 iterations SR reaches diversity values above 9800, while GA takes more than 10,000 generations to reach such degree of diversity.

The computational complexity of SR is larger than that of SA and GA. For SR the computation of forces and movements in SR grows with the square of the number of test data in the set. It also grows linearly with the number of test sets (NTS) and with the number of input variables in the test data (NIV). For both GA and SA, on the other hand, computational cost grows only linearly with those parameters. Even with this higher complexity, the great effectiveness of SR makes the test data generation process faster than with SA and GA and not so expensive. For instance, for \(|T| = 100\) (test sets in Fig. 2) SR takes about 200 iterations and 0.8 s to improve diversity from 522 (RTS) to 1063 (DOTS).

Because of this higher effectiveness we selected the SR algorithm to perform further evaluations of DOTG-ID. SA and GA could also be used and would likely lead to similar results with respect to those obtained with SR.

The additional evaluations are focused on two aspects: (1) how different factors affect the effectiveness of DOTG-ID, and (2) how effective DOTG-ID is for testing software. For evaluating the first aspect we carried out a Monte-Carlo simulation in which we studied the factors: test set size; type of failure pattern, defined by the shape of the failure pattern; and the failure size, determined by the relative size of the failure region with respect to the size of the program input domain \((D)\). For evaluating the second aspect we applied the technique for testing simple programs and evaluated the attained testing effectiveness in terms of mutation score and data-flow coverage. Standard random testing was used as a baseline for these evaluations.

### 5.1. A simulation study of factors influencing effectiveness

As discussed on Section 2.2, a failure region is a subset of the program input domain in which the software produces a faulty result [66,72]. For evaluation purposes one can simulate the existence of a failure region in a program by defining

<table>
<thead>
<tr>
<th>Table 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations and attained diversity for SA, GA and SR.</td>
</tr>
<tr>
<td>50</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>4000</td>
</tr>
</tbody>
</table>
a subset of the input domain for which all test data selected from the subset would “cause a failure”. Different failure patterns were proposed trying to capture typical shapes of failure regions related to program faults. As an instance, three programs with faults are presented by Chen et al. [14], each program has a fault such that their failure regions are a block (such a 2-dimensional square); a strip (a thin region); or a point (one or more small circular regions).

We used a 2-dimensional square floating point input domain. Values in each dimension can range from 0.0 to 100.0. Types of failure patterns were block, strip and point, similar to those presented in other studies [13–17,49].

Failure rate values ($\theta$) are given by the ratio between the area of the failure region and the area of the entire domain. As an instance, if a failure region is a $10.0 \times 10.0$ block on a $100.0 \times 100.0$ input domain, $\theta = 0.01$.

For the block pattern, an anchor point is randomly selected in the domain and a square is defined, centered on the point, with a size corresponding to the desired failure rate. For the strip pattern two anchor points are randomly selected on two adjacent sides of the domain with a thin stripe connecting them. The width of the stripe is calculated based on the desired failure rate. Points too close to the borders are avoided, because they would lead to a part of the strip being out of the

![Graphs showing effectiveness of DOTS and RTS](image)

Fig. 4. Effectiveness of DOTS and RTS ($\theta = 0.002 – 4(a)$; $0.01 – 4(b)$; and $0.05 – 4(c)$).
domain. Point failure patterns are created at 10 random anchor points in the domain. Equal sized radii for circles are defined based on the desired failure rate. Overlapped circles are not allowed. For block and point patterns, if any part of the failure region would be placed out of the domain, an adjustment to the size of the region is done to keep the desired failure rate.

Failure rate values ($\theta$) are 0.002; 0.01 and 0.05. For each failure rate we exploit several sizes for the test set ($|T|$) with orders of magnitude defined to produce a “reasonable” chance of finding the failure (e.g., for $\theta = 0.01$, $|T|$ ranges from 20 to 240 with steps of size 20).

The evaluation procedure is the following: for each failure rate $\theta$ and test set size $|T|$ we perform 2000 runs. For each run we generate a random test set (RTS) and evolve its diversity towards a DOTS using the SR algorithm. The DOTSs are evaluated for the three types of failure pattern. For each test set, different failure regions are generated, as described above. A counter is assigned to each type of failure pattern and is incremented if any test datum of the test set is placed inside the failure region. After all runs the counters’ values are used to compute the effectiveness measure of DOTS with respect to each failure pattern and each test set size. The SR parameters were: initial charge ranging from 1000 to 10,000; limit of iterations 600; and discharge coefficient ranging from 0.95 to 0.99.

We selected as effectiveness measure the number of the generated test sets that find the failure region, out of the total number of test sets. This value is computed for each test set size and failure rate and can range from zero, if no DOTS find the
failure region, to one if all 2000 DOTS do so. We also computed the ratio of effectiveness between the DOTS and the RTS (the ratio between the number of DOTS which find failures and the number of RTS which find failures). This ratio makes more visible the situations when one technique outperforms the other. The expected effectiveness of random testing in all cases is computed by using:

\[
ER(\theta, T) = (1 - (1 - \theta)^T) \times 2000.
\]  

5.1.1. The simulation results

Fig. 4a, b and c show the effectiveness of DOTS for each type of failure pattern and the expected effectiveness of random testing (Eq. (8)) for several test set sizes. Results are for values of \( \theta \) equal to 0.002, 0.01, and 0.05, respectively. The ratio of effectiveness between the DOTS and the RTS is also shown in Fig. 5a–c for the same values of \( \theta \).

The figures show that for the block failure pattern DOTS has better effectiveness than RTS. This advantage is higher for smaller failures (for \( \theta = 0.002 \) and 0.01) and for average test set sizes. For very small test sets RTS was equivalent to DOTS. As the test sets become very large RTS and DOTS tend to be closer in effectiveness. For the stripe failure pattern, results are similar, but the differences between RTS and DOTS are smaller. For the point failure pattern RTS is more effective than DOTS, which is equivalent to RTS only for the smaller failure rates (\( \theta = 0.002 \)).

Relation between effectiveness of DOTS and the failure pattern: clearly the differences between RTS and DOTS are more significant for the block pattern. For the stripe pattern, DOTS can be more effective than RTS for small failures and test sets that are not too small. For point pattern, RTS is more effective than DOTS except for small failures and for large test sets, situations in which the techniques are equivalent.

Fig. 6 shows the effectiveness of DOTS and RTS (\( \theta = 0.01 \)) for the block failure pattern, plotted with the average distances between adjacent test data in \( D \). These values are obtained by taking the test points in DOTS and using:

\[
AvgDist(T) = \text{Div}(T)/|T|.
\]  

This failure region is a 10.0 \( \times \) 10.0 block; when the value of \( \text{AvgDist}(T) \) becomes below 10.0 the effectiveness of DOTS goes close to 1.0. For instance, the ratio of effectiveness (between DOTS and RTS) is maximum for \( |T| = 120 \). For this test set \( \text{Div}(T) = 1154 \), so \( \text{Div}(T)/|T| = 9.62 \). It is the sieve effect: the test set forms a multi-dimensional sieve whose granularity, controlled by \( |T| \), determines the expected size of failure regions filtered in the program input domain.

Relation between effectiveness of DOTS and test set size: we observed that for very small \( |T| \) DOTS places a disproportionately large amount of test data at the boundaries of \( D \). This skin effect makes RTS more effective when using very small test sets. In this context the good sizes for the test sets can be based on \( \theta \); e.g., for \( \theta = 0.01 \) a good \( |T| \) is about 100. If \( |T| \) is too large a saturation effect takes place. For example, for \( \theta = 0.01 \) and \( |T| = 180 \) all DOTS find the failure region (effectiveness \( = 1.0 \)); beyond this size, DOTS stays at this maximum effectiveness, while RTS still improves (for \( \theta = 0.01 \) and \( |T| = 180 \) the RTS effectiveness is 0.8).

Note that the skin effect reduces the effectiveness of DOTS under the adopted assumption of failure regions randomly located in the input domain. With the assumption that “failures hide in corners” (not used in the simulation) the skin effect would play in favor of DOTS, instead of against it.

Relation between effectiveness of DOTS and failure size: the ratio of effectiveness between DOTS and RTS increases as \( \theta \) decreases. This effect happens mainly for the block pattern. Again, this is due to the skin effect: for smaller failures, the size of test sets used is larger, thus nullifying this effect.

Fig. 7 summarizes the ratio of effectiveness between DOTS and RTS for each type of failure pattern and for each failure rate \( \theta \). Values of \( |T| \) that are reasonable to be used considering \( \theta \) are \( |T| = 500 \) for \( \theta = 0.002 \); \( |T| = 100 \) for \( \theta = 0.01 \); and \( |T| = 40 \) for \( \theta = 0.05 \). It shows that the advantage of DOTS over RTS is larger for smaller failures.
5.2. Effect of diversity on data-flow coverage and mutation score

In this analysis a small set of programs were selected for the evaluation of the influence of the diversity values of the test sets on their effectiveness. We use data flow and mutation testing techniques for the analysis.

Data flow testing is a technique which selects required elements to be exercised based on the program’s data flow analysis. Points at which values are assigned to the program variables (definitions) and points at which values are recovered from program variables (uses) establish data flow associations and data flow paths to be covered in software testing. Under this perspective the quality of a test set is measured by the data flow coverage attained by the test set [65]. Mutation testing is a technique in which simple faults are introduced into the program, creating different versions of the program, each one with one seeded fault (the mutants). The quality of a test set is measured by its ability to distinguish the behavior of the mutants from the behavior of the original program [24].

The effectiveness for uncovering real program faults is not studied in this work. Instead, measures of data-flow coverage and mutation scores attained by the test sets are determined. These measures do not guarantee the technique’s effectiveness; they only give clues to the effects of improving the diversity of test sets. In any case, coverage measures, especially the coverage of faults in the mutation analysis, have been regarded as a reasonable estimation of the test set quality for real faults [4].

5.2.1. Evaluation procedure

We evaluate the random test sets (RTS) and the diversity oriented test sets (DOTS) through the following procedure:

**Preparation**
1. Select subject programs and values for test size ($NTD$).
2. For each program: identify the program interface, implement the driver to accept the test data from SR, and execute them in the subject program.
3. Define the number of test sets to be used ($NTS$). For this evaluation $NTS = 10$.

**Execution**
For each program:
   For each test size $NTD$:
1. Generate $NTS$ RTSs of size $NTD$. RTSs are generated using a uniform random number generator (with replacement) implemented in C and integrated into SR.
2. For each RTS apply the SR algorithm for generating $NTS$ DOTSs.

**Evaluation**
For each $NTS$ RTS and for each $NTS$ DOTS:
3. Using Poke-Tool [12] evaluate the data-flow coverage with respect to All-uses criterion and with respect to All-du-paths criterion.
5. For each program, each test set size and each testing criterion compute average value for diversity of test sets and the average and standard deviation values for coverage.
6. For each program, each test set size and each testing criterion perform statistical analysis on the coverage results.

5.2.2. Programs and test drivers

We used for the analysis three programs also used for evaluation purposes by other authors. The tritype program accepts three integers representing the lengths of the sides of a triangle, classifies the type of triangle and calculates its area [56]. The
expint program accepts an integer \( n \) and a floating point variable \( x \). It computes the real exponential integral of \( x \) \([63]\). The tcas program is part of a Traffic Alert/Collision Avoidance System that accepts information about altitudes of flights and produces safety warnings\(^3\).

To test programs it may be required to implement a test driver that receives input values from the SR algorithm (a sequence of integer or float numbers that defines the test data) and assigns the values to a data structure accepted by the program under test. For instance, to test a program that accepts as input a square three-dimensional matrix of integers, the testing driver simply accepts nine integers from the SR algorithm, assigns the values to the matrix positions, and calls the program passing the matrix (or a pointer to it) as a parameter.

5.2.3. Data flow and mutation coverage values

Tables 2, 4, 6 and 8 in this section summarize the results of the evaluation performed with the procedure in Section 5.2.1. For each test set size \( T \) we present the computed diversity value for the test set \( (Div) \), the coverage attained for the All-uses data flow criterion \( (Uses) \), the coverage with respect to the All-definition-use-paths criterion \( (DuP) \), and the attained mutation score using the fault-based criterion mutation analysis \( (Mut) \). Coverage values for All-uses and all-definition-use-paths represent the percentage value of the covered required elements. Coverage for mutation analysis represents the fraction of killed mutants. We do not perform an analysis to determine if the non-exercised testing requirements are feasible or not, such as infeasible def-use associations, infeasible du-paths or equivalent mutants. All values in the tables are obtained by calculating average values \( (\text{avg}) \) from 10 instances of test sets \( (NTS) \) for each different test set size \( (T) \) and for each different technique \( (RT \) or DOTG). For each coverage value the table also presents the standard deviation \( (\text{std}) \) calculated for the 10 instances.

For each table of coverage results we present a table of statistical information on these results \( (\text{Tables } 3, 5, 7 \text{ and } 9) \). The analysis is performed for each pair of coverage values \( \text{attained by RTS and by DOTS} \), for each test criterion \( (Uses, DuP \text{ and } Mut) \), and each test set size \( (T) \). The statistical analysis aims to indicate whether the differences on the coverage results attained by the techniques and test sets \( (RTS \text{ and DOTS}) \) are statistically significant, providing more reliable evidences on the relative effectiveness of the techniques \([43,67]\).

For each pair of coverage results of RTS and DOTS we applied a parametric hypothesis test for the difference between average values of two independent samples \( (t\text{-test}) \) and report the obtained \( p\text{-values} \). The null hypothesis is that the average results for coverage of RTS and DOTS are the same \( (H_0: \mu_{RTS} = \mu_{DOTS}) \), and the alternative hypothesis states that the average results for coverage are different \( (\mu_{RTS} \neq \mu_{DOTS}) \) and, therefore one of the techniques is more effective than the other.

5.2.3.1. Program expint

The interface of expint is defined as the integer value \( n \) ranging from 0 to 120 and the floating point value \( x \) ranging from 0 to 120, inclusive. These limits were defined by a brief analysis of the program specification and algorithm. A driver is used to map the values of the variable \( x \) handled by SR (floating point from 0 to 120) to values from 0 to 1.20 by dividing the original values obtained from the SR by 100 and passing the result as input to the program under test.

Components covered in DOTS but missed by RTS are related to the computation of extreme conditions \( n = 0 \text{ and } x = 0 \) and also are in the section of the algorithm that computes outputs when \( x \leq 1 \). This section has two nested loops and two selection commands whose conditions are indirectly influenced by the values of \( x \). The spread of values for \( x \) at this point appears

\(^3\) Program and documentation obtained from [http://www-static.cc.gatech.edu/aristotle/](http://www-static.cc.gatech.edu/aristotle/).
to be responsible for the coverage of some additional components. Table 2 shows results on the coverage attained by DOTS and RTS. Table 3 presents statistical information (p-values); each p-value refers to a pair of coverage values in expint (attained by RTS and by DOTS), for each test criterion (Uses, DuP and Mut), and each test set size (T). Statistical results for all programs are discussed in Section 5.2.4.

5.2.3.2. Program tritype. The interface of tritype is defined such that a, b, c are integers values ranging from 0 to 50. Because DOTS combine a range of values for the lengths of the sides a, b and c these test sets cover testing requirements related to both acute and obtuse scalene triangles. RTS often fails to cover one of these triangles in unlucky random attempts. The combination of extreme values presented in DOTS exercises the identification of some invalid triangles (including 0,0,0). Both RTS and DOTS, however, fail to cover testing requirements related to right triangles, which define a relatively small portion of the input domain. Table 4 shows results on the coverage attained by DOTS and RTS. Table 5 presents statistical information (p-values).

5.2.3.3. Program tcas. The tcas program accepts as inputs 15 variables. Six integer variables refer to absolute altitude values for planes and relative altitudes that separate planes. Five Boolean variables are parameters that control the kind of response the algorithm returns. One variable is a flag that can hold values 0, 1, 2, and 3. tcas has two major procedures: the main procedure and the alt-sep-test procedure, called by main.

For this program we decided to set the altitude variables ranging from 0 to 800. The other variables (Booleans and flag) are also represented as integer values in the same range. A test driver decodes the inputs from the SR domain to the tcas domain. This driver simply receives the values ranging from 0 to 800 and generates Boolean values: 0 for inputs from 0 to 400 and 1 for inputs from 400 to 800. The flag value is handled in a similar way. This kind of mapping creates a balanced

| Table 4 |
| Tritype – coverage results for DOTS and RTS. |
| T | Random test sets | Diversity-oriented test sets |
| | Div | Uses | DuP | Mut | Div | Uses | DuP | Mut | Div | Uses | DuP | Mut |
| 30 | 305.1 | 47.2 | 21.54 | 44.9 | 19.20 | 0.43 | 0.35 | 599.9 | 57.9 | 7.67 | 50.7 | 7.49 | 0.53 | 0.12 |
| 64 | 495.3 | 63.1 | 10.0 | 61.4 | 8.87 | 0.54 | 0.23 | 922.4 | 83.7 | 6.28 | 76.4 | 8.45 | 0.65 | 0.08 |
| 100 | 627.9 | 70.7 | 9.37 | 68.9 | 8.75 | 0.65 | 0.11 | 1194.9 | 83.7 | 5.76 | 77.4 | 7.90 | 0.79 | 0.08 |
| 150 | 832.1 | 70.7 | 8.22 | 68.9 | 7.23 | 0.65 | 0.10 | 1538.8 | 83.7 | 3.02 | 77.4 | 5.02 | 0.85 | 0.06 |

| Table 5 |
| Tritype – statistical results (p-values). |
| T | Uses | DuP | Mut |
| | | | |
| 30 | 0.064200 | 0.188960 | 0.024820 |
| 64 | 0.000280 | 0.000860 | 0.000260 |
| 100 | 0.001405 | 0.060355 | 0.000475 |
| 150 | 0.001243 | 0.034002 | 0.000310 |

| Table 6 |
| tcas-main-coverage results for DOTS and RTS. |
| T | Random test sets | Diversity-oriented test sets |
| | Div | Uses | DuP | Mut | Div | Uses | DuP | Mut | Div | Uses | DuP | Mut |
| 50 | 34,053 | 95.1 | 0 | 41.9 | 1.62 | 51,282 | 95.1 | 0 | 42.1 | 1.42 |
| 100 | 63,704 | 95.1 | 0 | 58.5 | 1.12 | 92,311 | 95.1 | 0 | 57.6 | 1.11 |
| 150 | 91,135 | 95.1 | 0 | 68.1 | 2.28 | 119,068 | 95.1 | 0 | 67.0 | 2.14 |

| Table 7 |
| tcas-main-statistical results (p-values). |
| T | Uses | DuP |
| | | | |
| 50 | 0.171965 | 0.377385 |
| 100 | 0.171965 | 0.058450 |
| 150 | 0.171965 | 0.146230 |
domain in SR which results in a balanced domain in the program input. In this way the mapping tends to transfer the diversity from the SR domain to the diversity of the input domain of the program under test. In fact, the effect of the mapping operations is to induce the combination of classes of values instead of the combination of the values themselves.

Results for \textit{tcas} are described in Tables 6 and 8. Table 6 shows coverage results for the main program and the diversity values for the test sets. Table 8 shows coverage values for the alt-sep-test procedure and mutation scores for both main and alt-sep-test. Tables 7 and 9 present statistical information (p-values).

For this program the results for procedure main were poorer than expected. In fact, the DOTS performed worse than RTS for \( T = 100 \) and \( T = 150 \). Coverage results for alt-sep-test indicate a better performance of DOTS compared to RTS in the All Uses and All Du-Paths coverage. The results concerning the mutation analysis criterion (for both main and alt-sep-test) indicate a slight advantage of DOTS over RTS.

The poor performance of DOTS for \textit{tcas} was a consequence of the definition of test set sizes smaller than necessary for the adequate coverage of the input domain. As the number of dimensions is greater than that of the previous programs the input domain is much larger, requiring a substantially larger amount of test data for thorough coverage. We observe that for this program DOTS explores the extreme points in the domain but leaves the middle of the domain with fewer testing points than RTS with the same size.

5.2.4. Summary

Table 10 summarizes the results found in Tables 2, 4, 6 and 8 and shows the effectiveness of DOTS when compared to RTS. Each value in Table 10 was obtained by using the following two-step process: (i) computing the ratio-program-criteria-size: the ratio between the coverage achieved by using DOTS and the coverage achieved by using RTS for each program, testing criteria, and test set size; and (ii) computing the ratio-program-criteria: the average of the values for ratio-program-criteria-size by using all test set sizes for each program and testing criteria.

Note that the ratio-program-criteria-size (step 1) measures how better (or worse) were DOTS when compared to RTS in a specific circumstance (a given program, testing criteria and test size). Step 2 computes the ratio-program-criteria and condenses this information by abstracting the test set sizes. The table shows that DOTS performed better than RTS in 10 out of 12 cases. In these 10 cases DOTS outperforms RTS by a ratio ranging from 1.032 (expint evaluated with All-Uses) up to 1.230 (tritype evaluated with Mutation Analysis).

Another important characteristic of DOTS can be seen by the analysis of the standard deviation values in Tables 2, 4, 6 and 8. Standard deviation values for DOTS are consistently lower than the values for RTS when we compared a similar scenario (given the program, a specific testing criterion and specified test set size). This can be seen as a good characteristic: a lower variability in the effectiveness when using different test sets can make the testing results more predictable.

A detailed analysis of Tables 2, 4, 6 and 8 shows a total of 39 pairs of coverage values (attained by RTS and by DOTS) for all programs, test criteria, and test set sizes. Out of these 39 pairs of values, in 35 pairs the coverage achieved by DOTS was

### Table 8
Coverage results for alt-sep-test and mutation scores for main and alt-sep-test.

<table>
<thead>
<tr>
<th></th>
<th>Random test sets</th>
<th>Diversity-oriented test sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>66.3</td>
<td>5.09</td>
</tr>
<tr>
<td>100</td>
<td>72.6</td>
<td>3.33</td>
</tr>
<tr>
<td>150</td>
<td>72.6</td>
<td>5.98</td>
</tr>
</tbody>
</table>

### Table 9
Alt-sep-test, main-statistical results (p-values).

<table>
<thead>
<tr>
<th></th>
<th>Uses</th>
<th>DuP</th>
<th>Mut</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.032420</td>
<td>0.010700</td>
<td>0.012025</td>
</tr>
<tr>
<td>100</td>
<td>0.045620</td>
<td>0.000310</td>
<td>0.009055</td>
</tr>
<tr>
<td>150</td>
<td>0.085065</td>
<td>0.085040</td>
<td>0.002795</td>
</tr>
</tbody>
</table>

### Table 10
Ratio between DOTS coverage and RTS.

<table>
<thead>
<tr>
<th>Program</th>
<th>Uses</th>
<th>DuP</th>
<th>Mut</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{Expint}</td>
<td>1.032</td>
<td>1.035</td>
<td>1.189</td>
</tr>
<tr>
<td>\textit{Tritype}</td>
<td>1.227</td>
<td>1.115</td>
<td>1.230</td>
</tr>
<tr>
<td>\textit{Tcas-main}</td>
<td>1.000</td>
<td>0.989</td>
<td>1.059</td>
</tr>
<tr>
<td>\textit{Tcas-alt-set-test}</td>
<td>1.064</td>
<td>1.118</td>
<td>1.059</td>
</tr>
</tbody>
</table>
greater than that achieved by RTS (cov(DOTS) > cov(RTS)). Only in four pairs of values the coverage achieved by DOTS was lower than that achieved by RTS (cov(DOTS) < cov(RTS)).

Regarding the statistical results, lower p-values indicate stronger evidences against $H_0$ and, therefore, more significant indication that one technique is really more effective than another [43]. For the analysis of the statistical tables (Tables 3, 5, 7 and 9) we need to distinguish two classes of results: class (i) is when cov(DOTS) > cov(RTS) and class (ii) is when cov(DOTS) < cov(RTS).

i. $\text{cov}(\text{DOTS}) \geq \text{cov}(\text{RTS})$: from these 35 pairs of values, in 17 the evidences are very strong ($p < 0.01$), in 7 the evidences are strong ($0.01 \leq p < 0.05$), in 4 the evidences are moderate ($0.05 \leq p < 0.10$), and in 7 the evidences are low ($p > 0.10$), that DOTS is more effective than RTS.

ii. $\text{cov}(\text{DOTS}) < \text{cov}(\text{RTS})$: from these 4 pairs of values, in one the evidences are moderate ($0.05 \leq p < 0.10$), and in 3 the evidences are low ($p > 0.10$), that RTS is more effective than DOTS.

The overall statistical results point out that DOTS is more effective than RTS, with strong or very strong evidences ($p < 0.05$), in 24 pair of coverage values. On the other side, only in one pair of coverage values RTS is more effective than DOTS, with moderate evidences ($0.05 \leq p < 0.10$).

6. Conclusions

We have presented a new testing technique, diversity oriented test data generation, which uses a concept named diversity of test sets for test data generation. We propose that this concept can be used to translate into an automatic test data generation technique the intuitive belief that the variety, or diversity, of the test data used to test a program can be positively related to the quality of the test performed (e.g., Whittaker and Voas say: "... testers should count inputs from every user and ensure that those inputs are diverse enough to form a complete set of test data" [76]).

The diversity is a measure that can be defined under different perspectives, using distinct types of information to determine diversity values for the test sets. In this work we explore the input domain perspective for diversity (DOTG-ID), which considers the distances among the test data in the program input domain to compute a diversity value for test sets.

We used the well-known metaheuristic search techniques, simulated annealing and genetic algorithm, and a new proposed heuristic, the simulated repulsion (SR), as optimization tools to generate the diversity oriented test sets (DOTS). Using SR we carried out empirical evaluations of DOTG-ID: a Monte Carlo simulation was performed to assess how different factors affect the effectiveness of the DOTG-ID; the technique was also applied for testing simple programs and the attained testing effectiveness in terms of mutation score and data-flow coverage was evaluated.

Our simulation indicates that the technique can improve the effectiveness of the standard random testing (up to 35%) and that this improvement appears to increase for failure regions with smaller failure rates. Our experiments with programs indicate that DOTG-ID can improve the attained coverage values, mainly the mutation score, in most cases (up to 30%, for tritype, $|T| = 150$). However, RTS can perform better for small values of $|T|$ with respect to the size of the input domain, as shown in the simulation and also in the coverage results for tcas.

We are currently working on different perspectives for DOTG (such as DOTG-SP, briefly described in Section 1) and developing extensions to DOTG-ID. These extensions focus on defining and evaluating DOTG-ID for software which accept as input complex information such as data structures (information systems) and programs (compilers).

Acknowledgments

This work was partially supported by CAPES (1544/01-2) and CNPq (380764/2005-9). We also thank all the reviewers, whose valuable comments helped us to improve the paper.

References


4 These cases are: $\text{expint}, T = 20$, criterion = uses; $\text{expint}, T = 50$, criterion = DuP; $\text{tcas-main}, T = 100$, criterion = DuP; and $\text{tcas-main}, T = 150$, criterion = DuP.


