Observations in using Parallel and Sequential Evolutionary Algorithms for Automatic Software Testing

Enrique Alba [*], Francisco Chicano

Grupo GISUM, Dept. de Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN

Abstract

In this paper we analyze the application of parallel and sequential evolutionary algorithms to the automatic test data generation problem. The problem consists of automatically creating a set of input data to test a program. This is a fundamental step in software development and a time consuming task in existing software companies. Canonical sequential evolutionary algorithms have been used in the past for this task. We explore here the use of parallel evolutionary algorithms. Evidence of greater efficiency, larger diversity maintenance, additional availability of memory/CPU, and multi-solution capabilities of the parallel approach, reinforce the importance of the advances in research with these algorithms. We describe in this work how canonical genetic algorithms (GAs) and evolutionary strategies (ESs) can help in software testing, and what the advantages are (if any) of using decentralized populations in these techniques. In addition, we study the influence of some parameters of the proposed test data generator in the results. For the experiments we use a large benchmark composed of twelve programs that includes fundamental algorithms in computer science.

Key words: Software testing, evolutionary algorithms, evolutionary testing, parallel evolutionary algorithms

[*] Corresponding author.
Email addresses: `eat@lcc.uma.es` (Enrique Alba), `chicano@lcc.uma.es` (Francisco Chicano).

1 Introduction

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know whether a software module fulfills a set of requirements (the specification). Modern software is very complex and these techniques have become a necessity in most software companies. One of these techniques is *formal verification*, in which some properties of the software can be checked much like a mathematical theorem defined on the source code. Two very well-known logics used in this verification are *predicate calculus* [1,2] and *Hoare logic* [3]. However, formal verification using logics is not fully automatic. Although automatic theorem provers can assist the process, human intervention is still needed. Another well-known and fully automatic formal method is *model checking* [4]. In this case all the possible program states are analyzed (in a direct or indirect way) in order to prove that the program satisfies a given property. This property is specified using a temporal logic like LTL [5] or CTL [6]. Both in formal verification and model checking a model of the program is required in order to prove (or refute) the properties we want to check. In order to ensure that the model correctly captures the behavior of the program, some refinement-based approach is needed along the development process. One of the best known model checkers is SPIN [7], which takes a software model codified in PROMELA (a programming language usually not used in real programs) and a property specified in LTL as input. The drawback of using the PROMELA language is currently solved by the use of translations tools such as the one integrated in the Bandera tool kit [8] or the one described in [9], which translates Java code into the intermediate language accepted by SAL model checker [10]. However, we can also find model checkers like Java PathFinder [11], which in its last versions directly works on bytecodes of multi-threaded Java programs. The main drawback of a model checking approach is the so-called state explosion: when the size of the program increases, the amount of required memory also increases but in an exponential way. This phenomenon limits the size of the models to be checked. Some techniques used to alleviate this problem are partial order reduction [12], symbolic model checking [13], and symmetry reduction [14].

Nevertheless, the most popular technique used to check software requirements is *software testing*. With this technique, the engineer selects a set of program inputs and tests the program with them. If the program behavior is as expected, s/he assumes that it is correct. Since the size of the input data set is the engineer's decision, s/he can control the effort dedicated to the testing task. This is a very important, time consuming, and hard task in software development [15–17]. *Automatic test data generation* consists of automatically proposing a suitable set of input data for a program to be tested (the *test program*). It comes as a way of releasing engineers from the task of finding the best set of inputs to check program correctness. The automatic generation of test data for computer programs has been dealt with in the literature for many years [18,19]. We can distinguish four large paradigms in search-based software testing that differ in the kind of information they use to generate the test data: structural testing, functional testing, grey-box testing, and non-functional testing [20].

In *structural testing* [21–25] the test data generator uses the structural information of the program to guide the search of new input data (for this reason it is also called white-box testing). Usually, this structural information is gathered from the *control flow graph* of the program. Structural testing seeks to execute every testable element under consideration (whether statements, branches, conditions, etc.), but can only detect faults where one of the executed elements produces obviously incorrect behavior. If no error is found the correctness of the program can not be assured. In *functional testing* [26,27] the information used by the test data generator is a kind of specification of the behavior of the program. The objective is to check that the software behaves exactly as specified using no information about the internal structure of the program (black-box testing). The paradigm known as *grey-box testing* [28] is a combination of the two previous ones (white-box and black-box testing). It uses structural and functional information in order to generate input data. For example, in assertion testing some asserts are introduced in the source code to check the functional behavior, while structural testing techniques are used in order to generate input data violating these assertions. Finally, in *non-functional testing* [29,30] the objective is to check any aspect of the program which is not related to its functional behavior. Some examples of non-functional attributes that could be checked are usability, portability, memory use, efficiency, etc.

Focusing on structural testing, which is the paradigm followed in this work, we can find several alternatives in the literature. In the so-called *random test data generation*, the test data are created randomly until the objective is satisfied or a maximum number of input data are generated [21,31]. *Symbolic test data generation* [18] involves using symbolic rather than concrete values in order to get a symbolic execution. Some algebraic constraints are obtained from this symbolic execution and these constraints are used for finding test cases [32]. A third (widespread) approach is *dynamic test data generation*. In this case, the program is instrumented to pass information to the test generator. The test generator checks whether the test adequacy criterion is fulfilled or not. If the criterion is not fulfilled it creates new test data to serve as input for the program. The test data generation process is translated into a function minimization problem, where the function is some kind of "distance" to a desirable execution where the test criterion is fulfilled. This paradigm was presented in [19] and much work has been based on it [21,30,33,34]. Some hybrid techniques combining symbolic and concrete execution have been explored with very good results. This is the case of the tools DART [25] and

CUTE [24].

Sticking to the dynamic test data generation paradigm, several metaheuristic techniques have been applied to the problem in the past. Mantere and Alander in [35] present a recent review on the application of evolutionary algorithms to software testing. Most of the papers included in their discussion use genetic algorithms (GAs) to find test data. In fact, only a few articles listed in the review include other techniques such as Cultural Algorithms [36] (a special kind of GA), Hill Climbing [37], and Simulated Annealing [38]. We have found other recent works applying metaheuristic algorithms to software testing. In [39] the authors explain how a Tabu Search algorithm can be used to generate test data obtaining maximum branch coverage. Sagarna and Lozano tackle the problem by using an Estimation of Distribution Algorithm (EDA) in [40], and they compare a Scatter Search (SS) with EDAs in [41].

In this work we analyze the application of several decentralized and centralized evolutionary algorithms (EAs) to the automatic test data generation problem, specifically genetic algorithms and evolutionary strategies (whose application to software testing was introduced by the authors for the first time in [42]). We describe how they can be applied to the software testing problem by analyzing several alternatives. In addition, we study the influence of some parameters of the proposed test data generator on the results. The rest of the paper is organized as follows. We detail the construction of our test data generator in the next section. Section 3 presents a general description of parallel and decentralized EAs. Then, in Section 4 we analyze the results obtained in the experimental study. Finally, Section 5 presents the final conclusions and future work.

2 The Test Data Generator

In this section we describe the proposed test data generator and the whole test data generation process. We must define a test adequacy criterion in order to formalize the objective of the generator, that is, we need a condition which any input data set should fulfill in order to be considered an adequate testing set. In this work we use the *condition coverage* test adequacy criterion. This criterion requires that all the atomic conditions of the test program be evaluated to the two boolean values: *true* and *false*. Other well-known test adequacy criteria are *branch coverage*, requiring all the branches to be taken, and *statement coverage*, in which all the program statements must be executed. It is important to note that the condition coverage criterion is harder than branch and statement coverage in the case of programs in C language. That is, if we find a set of input data that makes all the atomic conditions of a C program take the possible boolean values, then we can ensure that all the feasible branches will be taken and, in consequence, all the reachable statements will be executed. However, the opposite is not true, i.e., executing all the reachable statements or taking all the feasible branches does not ensure that all the atomic conditions will take the feasible boolean values. This fact makes condition coverage equivalent to *condition-decision coverage* [21] in C programs and this is the reason why it was selected here.

Our test data generator breaks down the global objective (the condition coverage) into several partial objectives consisting of making one atomic condition take one boolean value. For example, from the fragment of the control flow graph seen in Fig. 1 we can extract six partial objectives: to make condition one true (`c1t`), to make condition one false (`c1f`), etc. Then, each partial objective can be treated as an optimization problem in which the function to be minimized is a distance between the current input and an input satisfying the partial objective. For this reason we call this function the *distance function*. In order to solve such minimization problem, global search techniques (evolutionary algorithms in our case) can be used.

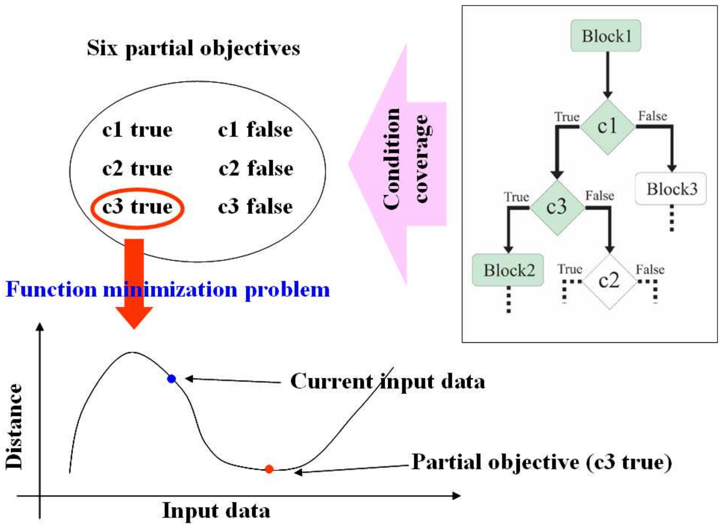

Fig. 1. We identify six partial objectives in this fragment of the control flow diagram.

2.1 Distance Function

Following on from the discussion in the previous section, we have to solve several minimization problems: one for each atomic condition and boolean value. The distance function to be minimized depends on the expression of the particular atomic condition and the values of the program variables when the condition is reached. In Table 1 we show the distance function for each kind of condition and each boolean value.

The distance function of a particular partial objective can only be computed if

Table 1
Distance functions for different kinds of conditions and boolean values. The variables a and b are numeric variables (integer or real)

Condition type	*true* expression	*false* expression
$a < b$	$a - b$	$b - a$
$a <= b$	$a - b$	$b - a$
$a == b$	$(b - a)^2$	$(1 + (b - a)^2)^{-1}$
$a != b$	$(1 + (b - a)^2)^{-1}$	$(b - a)^2$
a	$(1 + a^2)^{-1}$	a^2

the program flow reaches the associated atomic condition, because it depends on the values of the program variables at that point of the program execution. For this reason, when the condition is not reached, the distance function takes the maximum possible value for a real number in a machine using 64-bit IEEE 754 representation (that is, $2^{1024} - 2^{971}$).

2.2 Program Instrumentation

We instrument the source code of the program in order to get information about the value of the distance function and the conditions traversed in a program run. The instrumentation must be done carefully to avoid a change in the program behavior. This step is performed automatically (not manually) by our application that parses the C source program and generates a modified C source program with the same original behavior. This application transforms each atomic condition into an expression that is evaluated to the same value as the original condition. This expression has a (neutral) side effect: it informs about the boolean value it takes, and the distance value related to the condition. If `<cond>` is an atomic condition in the original program, the associated expression used instead of the original condition in the modified program is:

```
((<cond>)?
    (inform(<ncond>,1),(distance(<ncond>,<true_expr>,<false_expr>),1)):
    (inform(<ncond>,0),(distance(<ncond>,<true_expr>,<false_expr>),0)))
```

where `<ncond>` is the number of the condition in the program, `<true_expr>` and `<false_expr>` are the fitness expressions for the true and false values of the condition, `inform` is a function that informs the test data generator about the condition reached and its value, and `distance` is a function that informs about the distance value. This transformation does not modify the functional behavior of the program unless the original atomic condition has side effects. [1]

In the evaluation of an input, when the test data generator executes the modified test program with such input, a report of the conditions reached and the distance values are computed and transmitted to the generator. With this information the generator builds a coverage table where it stores, for each condition, the set of test data that makes the condition true and false throughout the process. That is, for each condition the table stores two sets: the true set and the false set. This table is an important internal data structure that is consulted during the test data generation. We say that a condition is *reached* if at least one of the sets associated with the condition is non-empty. On the other hand, we say that a condition is *covered* if the two sets are non-empty.

2.3 Test Data Generation Process

Once we have presented the distance functions and the instrumentation details we can now focus on the test data generator itself. The main loop of the test data generator is shown in Fig. 2.

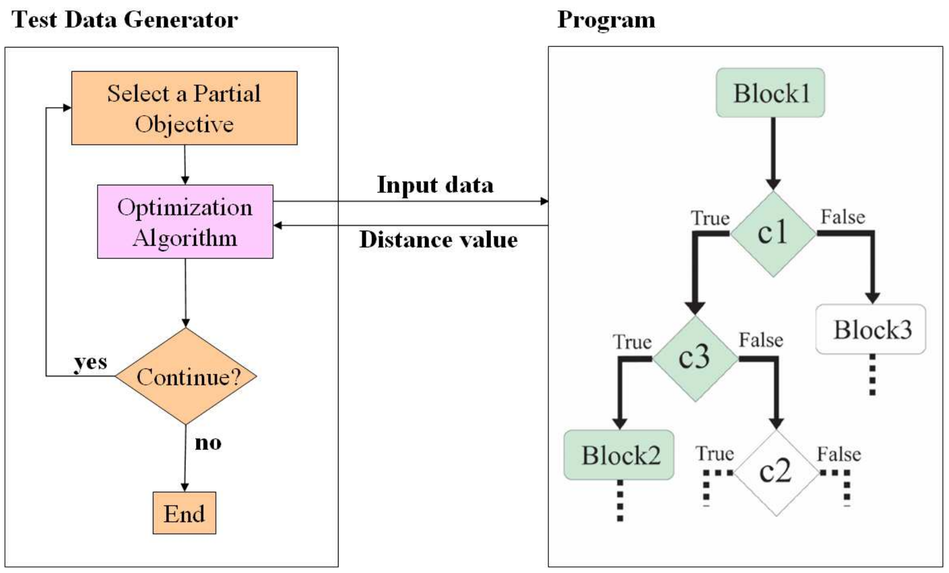

Fig. 2. The test data generation process.

At the beginning of the generation process some random inputs (10 in this work) are generated in order to reach some conditions. Then, the main loop of the generator begins and the generator selects a partial objective not covered. The test data generator does not select the partial objectives in a random way.

[1] This can be explained because the fitness expressions are formed by combining the operands of the atomic condition.

As we said before, the distance function of each partial objective depends on the expression of the particular atomic condition and the values of the program variables when the condition is reached. This means that the distance can only be computed if the program flow reaches the atomic condition. Thus, the test data generator always selects a partial objective with an associated condition reached by a previous input.

When the partial objective is selected, it uses the optimization algorithm to search for test data making that condition take the value not yet covered. The optimization algorithm is seeded with at least one input reaching the mentioned condition. The algorithm tries different input data and uses the distance value to guide the search. During this search test data covering other conditions can be found. These test data are also used for updating the condition table. In fact, we can set the stop condition of the optimization algorithm to cover a partial objective not yet covered (we study this alternative in the experimental section since it is not our main approach here). As a result, the optimization algorithm can or can not find a solution. In any case, when the algorithm stops, the main loop starts again and the test generator selects a different partial objective. This scheme is repeated until a total condition coverage is obtained or a maximum number of consecutive failures of the optimization algorithm is reached. When this happens the test data generator exits the main loop and stops.

When we use a decentralized optimization algorithm, such as a distributed evolutionary algorithm, we have several subalgorithms working independently of each other with some sparse interconnections among them. In this case we can assign a different partial objective to each subalgorithm. If all the partial objectives are solved approximately at the same time, the search could be accelerated. This alternative will also be analyzed in the course of this article.

2.4 Coverage Metrics

In order to end the description of the test data generator we must discuss the coverage metric used to report the generator results. The simplest coverage metric is the ratio between the covered partial objectives and the total number of partial objectives which, expressed as a percentage, gives the *coverage percentage* (condition coverage percentage in our case).

Although the coverage percentage is the simplest way of reporting generator efficacy, it is not the more appropriate metric. This is because it is impossible for some programs to reach total coverage, because of the presence of unreachable partial objectives. In this case a global coverage loss is produced irrespectively of the techniques used for testing. For example, an infinite loop

```
while(1)
{
/* The previous condition
is always true */
⋮
}
```

```
char *a;
p = (char *)malloc (4);
if (!p)
{
        fprintf("Error");
        exit(0);
}
```

Fig. 3. Two pieces of code that prevent a program from reaching 100% of condition coverage. The left one produces a code-dependent coverage loss, while the right one produces an environment-dependent coverage loss.

has a condition that is always true and never false (Fig. 3 left). Another example is the condition `(sign(x)>2)`, where the function `sign` can only return three values: -1, 0, +1. In these cases there are pairs (`condition`, `boolean`) that are *unreachable* and no test data generator can reach 100% of condition coverage due to the test program itself. In this situation we refer to a *code-dependent coverage loss*. However, there is another factor which may produce an unavoidable coverage loss: the environment in which the program is executed. One example of this situation is related to dynamic memory allocation. Suppose that a program allocates some dynamic memory and then checks if this allocation has failed. Most probably it succeeds for all the tests run on the program, and the check condition gets only one value. In this case we talk about an *environment-dependent coverage loss* (Fig. 3 right). When one of these situations appears in a program no test data generator can get total coverage and it may appear to be ineffective when, in fact, it is not. For example, we can get a low coverage percentage in one program but this low percentage might happen to be the maximum coverage that it is possible to obtain.

We search for a coverage metric taking into account the coverage loss, a measurement that gets its known maximum value when it is impossible to get new input data covering more partial objectives. For this reason we have introduced another metric that we call the *corrected coverage* and that is computed as the ratio between the number of covered and reachable partial objectives. In this metric the unreachable partial objectives are not taken into account, without any loss of information or drawback for the testing task. This measure is useful for comparing the performance of the test data generator in the different programs. In this way, we can sort the programs in order of difficulty. If we use the simple condition coverage to compare the programs we can classify a program as difficult when it has a lot of unreachable partial objectives but the remaining partial objectives can be covered easily. However, the computation of the corrected coverage requires knowing the unreachable partial objectives. In small programs these partial objectives can be easily determined but it can be a difficult task in larger programs (it could be a NP-hard problem itself).

In these cases the corrected coverage measure is not practical. In this work, we decide by human observation whether a partial objective is reachable or not and the unreachable partial objectives are then communicated to the test data generator via configuration files. We use the corrected coverage in the experiments in order to avoid code-dependent coverage loss. The environment-dependent coverage loss is more difficult to avoid and the new metric does not take this loss into account.

At this point we need to modify the way in which the partial objective selection is performed at the beginning of the main loop of the test data generator in order to count the correct number of evaluations required for the coverage measured. As a matter of fact, in order to avoid unnecessary work the unreachable partial objectives should not be taken into account in the partial objective selection.

3 Evolutionary Algorithms

In this section we intend to provide a quick overview of the evolutionary algorithm family (sequential and parallel) in order to classify and explain the algorithms we are using in the paper.

Evolutionary Algorithms (EAs) [43] are metaheuristic search techniques loosely based on the principles of natural evolution, namely adaptation and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. They are based on a set of tentative solutions (individuals) called *population*. The problem knowledge is usually enclosed in a function, the so-called *fitness function*, which assigns a quality value to the individuals. It is usual for many EA families to manipulate the population as a single pool of individuals. By *manipulation* we mean applying a selection of the fittest individuals, recombination of slices of two individuals in order to yield one or two new children, and mutation of their contents. Frequently, EAs use these variation operators in conjunction with associated probabilities that govern their application at each step of the algorithm. In general, any individual can potentially mate with any other by applying a centralized selection operator. The same holds for the replacement operator, where any individual can potentially leave the pool and be replaced by a new one. This is called a *panmictic* population of individuals. A different (decentralized) selection model exists in which individuals are arranged spatially, therefore giving rise to *structured EAs*. Most other operators, such as recombination or mutation, can be readily applied to either of these two models [44]. Centralized versions of selection are typically found in serial EAs, although some parallel implementations also use it. For example, the *global parallelism* approach evaluates in parallel the individuals of the population (sometimes recombination and/or mutation are also parallelized), while still using a centralized selection performed sequentially in the main processor guiding the base algorithm [45]. This algorithm keeps the same behavior as the sequential centralized one, although it performs time-consuming objective functions much faster.

However, a great amount of parallel EAs found in the literature normally utilize some kind of spatial disposition for the individuals, and then parallelize the resulting chunks in a pool of processors. Decentralizing a single population can be achieved by partitioning it into several subpopulations, where island EAs are run performing sparse exchanges of individuals (distributed EAs), or in the form of neighborhoods (cellular EAs). In distributed EAs, additional parameters controlling when migration occurs and how migrants are selected/incorporated from/to the source/target islands are needed [46,47]. In cellular EAs, the existence of overlapped small neighborhoods helps in exploring the search space [48]. These two kinds of EAs seem to provide better sampling of the search space and seem to improve the numerical and runtime behavior of the basic algorithm in many cases [49,50].

In the present study we mainly focus on distributed EAs. This model can be readily implemented in a cluster of workstations, which is one main reason for its popularity. A migration policy controls the search. The migration policy must define the island topology, when migration occurs, which individuals are being exchanged, what type of synchronization among the subpopulations is used, and the kind of integration of exchanged individuals within the target subpopulations. The advantages of a distributed model (either running on separate processors or not) is that it is usually numerically faster than a panmictic EA. The reason for this is that both the run time and the number of evaluations are potentially reduced thanks to its separate search from several regions of the problem space. High diversity and species formation are two of the best reported features of distributed EAs.

After this general introduction to parallel and decentralized evolutionary algorithms in the next sections we focus on the details of the specific evolutionary algorithms used in this work to tackle the software testing problem. We find four well-established main types of EA in the literature: Genetic Algorithm (GA) [51], Evolutionary Strategy (ES) [52], Evolutionary Programming (EP) [53], and Genetic Programming (GP) [54]. The main differences between the four kinds of algorithms reside in the representation of the solutions and the variation operators used. In this work we use Evolutionary Strategies and Genetic Algorithms.

3.1 Evolutionary Strategies

The Evolutionary Strategy algorithm was introduced for software testing in a recent work [42]. In an ES [52] each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations (σ) and, optionally, a vector of angles (ω). These two last vectors are used as parameters of the main Gaussian mutation operator used by this technique. The additional parameters evolve together with the program variables, to allow the search strategy to adapt itself to the landscape of the search space.

For the selection operator many alternatives exist: roulette wheel, random selection, q-tournament selection, and so on (see [55] for a good review of selection operators). All the selection operators can be used with any EA since the only information needed to select the individuals is the fitness value (the selection operator does not depend on the representation used).

In the recombination operator of an ES each of the three real vectors of an individual can be recombined in a different way. Some of the several possibilities for the recombination of the real vectors are discrete, intermediate, and generalized intermediate recombination [43]. The recombination operator is not so important as the mutation. In fact, we do not use recombination in our algorithm.

The mutation operator is governed by the three following equations:

$$\sigma'_i = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \ , \tag{1}$$
$$\omega'_i = \omega_i + \varphi N_i(0,1) \ , \tag{2}$$
$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \ , \tag{3}$$

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω', $N(0,1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix C. The subindex i in the standard normal distribution indicates that a new random number is generated anew for each component of the vector. The notation $N(0,1)$ is used to indicate that the same random number is used for all the components. The parameters τ, η, and φ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [56].

With respect to the replacement operator, there is a special notation to indicate whether the old population is taken into account or not to form the new population. When only the new individuals are used, we have a (μ, λ) replacement; otherwise, we have a $(\mu + \lambda)$ replacement, where the best μ individuals from the union of the old population and the offspring are selected to form the new population. As it was the case in the selection operator, the replacement operator only needs the fitness value of the individuals to form the new population. Thus, we can use the two mentioned operators in any kind of EA.

3.2 Genetic Algorithms

The first work applying Genetic Algorithms to software testing was published in 1992 [57] and the first in depth study of their capabilities in the software testing domain was performed in the PhD. Thesis of Sthamer [58]. A Genetic Algorithm is an EA that usually represents the solutions by means of a binary string (chromosome). However, other representations have been used with GAs, so we can not use this feature as a distinguishing one. In fact, the individuals we use in our GA are vectors of numeric values and they can be integers (`int` type) or reals (`double` type). We use this representation in order to avoid precision loss when using a low number of bits for representing each numeric input of the test program.

Unlike the ES, the recombination operator is of a great importance in GAs. Some of the most popular recombination operators are the single point crossover (SPX), the double point crossover (DPX), and the uniform crossover (UX) [59]. The first one selects a random position in the chromosome and the two parents exchange their slices located before and after this point to generate two offspring. In DPX, two points are stochastically defined and the two parents exchange the slices bounded by these two limits creating again two offspring (however, we select only one in our experiments). Finally, in UX each bit of the offspring is randomly selected from the two parents. All these recombination operators can also be applied to our particular representation by replacing the bits by vector components.

The traditional mutation operator works on binary strings by probabilistically changing every position to its complementary value. However, for representations based on vectors of numeric values (like the one used here) the mutation could add some random value to the components of the vector. In our case, the probability distribution of these random values is a normal distribution with mean 0. The standard deviation of the distribution is a parameter of the mutation operator and we reveal its value in the experimental section.

At this point we can talk about the differences between ES and GA. In general, ES is characterized by the structure of the individuals (including at least one separated vector of self-adaptive parameters for the mutation) and the mutation operator. This is the main difference between the two algorithms. Unfortunately, we can not give a clear rule to characterize GAs because there is considerable variance in the literature in the field. However, the GAs usually do not have self-adaptive parameters or the self-adaptation mechanism is different from that of the ES. For this reason we conclude that the main difference between the algorithms focuses on the self-adaptation.

4 Experiments

In this section we present the experiments performed over a benchmark of twelve test programs in C covering some practical and fundamental aspects of computer science. The programs range from numerical computation (such as Bessel functions) to general optimization methods (such as simulated annealing). Most of the source codes have been extracted from the book "C Recipes" available on-line at `http://www.library.cornell.edu/nr/bookcpdf.html`. The programs are listed in Table 2, where we present information on the number of conditions, the lines of code (LOC), the number of input arguments, a brief description of their goal, and how they are accessed.

Table 2
Programs tested in the experiments. The source column presents the name of the function in C-Recipes

Program	**Conds.**	**LOC**	**Args.**	**Description**	**Source**
`triangle`	21	53	3	Classify triangles	Ref. [21]
`gcd`	5	38	2	Greatest Common Denominator	Authors
`calday`	11	72	3	Calculate the day of the week	`julday`
`crc`	9	82	13	Cyclic Redundant Code	`icrc`
`insertion`	5	47	20	Sort by insertion method	`piksrt`
`shell`	7	58	20	Sort by shell method	`shell`
`quicksort`	18	143	20	Sort by quicksort method	`sort`
`heapsort`	10	72	20	Sort by heapsort method	`hpsort`
`select`	28	200	21	kth element of an unordered list	`selip`
`bessel`	21	245	2	Bessel J_n and Y_n functions	`bessj*,bessy*`
`sa`	30	332	23	Simulated Annealing	`anneal`
`netflow`	55	112	66	Network optimization	Wegener [60]

The first test program, `triangle`, receives three integer numbers as input and decides what kind of triangle they represent: equilateral, isosceles, scalene, or no triangle. The next program, `gcd`, computes the greatest common denominator of the two integer arguments. The `calday` test program computes the day of the week, given a date as three integer arguments. In `crc` the cyclic redundant code is computed from 13 integer numbers given in the arguments. The next four test programs (`insertion`, `shell`, `quicksort`, and `heapsort`) sort an input list with 20 real arguments using well-known sorting algorithms. The `select` program gets the kth element from an unsorted list of real numbers. The first argument is k and the rest of the arguments form the unsorted list. The next program computes the Bessel functions given an order n and a real argument. The `sa` program solves an instance of the Travelling Salesman Problem with ten cities by using Simulated Annealing. The first three arguments are seed parameters for the pseudorandom number generator and the rest are 20 real numbers representing the two-dimensional position of each city. Finally, the `netflow` program optimizes a net architecture for maximum data flow. The input is the description of the net to be optimized. Its size is limited to 6 nodes and 10 edges.

In the following section we discuss the representation and the fitness function used in the algorithms. In Section 4.2 we present the first results comparing the parallel decentralized and the sequential centralized versions of the EAs used. The next section compares the ES-based test data generators against the GA-based ones. Then, we perform a study of the influence in the results of several parameters of the dES-based test data generator. Finally, in Section 4.5 we compare our best algorithm against others found in the literature.

4.1 Representation and Fitness Function

As stated before, the input values of the test programs used in the experiments are real or integer numbers. In the ES-based algorithms each of these numeric values is represented with a real value in the solution vector of the individual. This value is rounded to the nearest integer in the case of integer arguments when the test program is executed. In the case of the GA-based algorithms the input values are directly mapped to the chromosome itself without any extra conversion. By using such unbounded numeric representation in both algorithm classes we can explore the whole solution space. This contrasts with other techniques that limit the domain of the input variables to a bounded region [40].

The fitness function used in the evolutionary search is not exactly the distance function. We want to avoid negative fitness values in order to be able to apply selection operators that depend on the absolute value of the fitness function such as the Roulette Wheel selection. For this reason we transform the distance value by using an `arctan` function that maps the whole real number set into a bounded interval. The resulting fitness function is:

$$fitness(\mathbf{x}) = \pi/2 - \arctan(distance(\mathbf{x})) + 0.1 \ . \tag{4}$$

In the previous expression we multiply the `arctan` result by -1 since our EA software is designed to maximize the fitness function. Thus, we need to add the value $\pi/2$ to the expression in order to always get a positive result. Finally, the 0.1 value is added so that negative values are not obtained when there is precision loss in the difference calculation.

4.2 Parallel Distributed vs Sequential Panmictic EAs

In this section we shall compare the results obtained by the test data generators with parallel decentralized EAs and sequential centralized EAs as the search engine. In particular, we compare the distributed ES (dES) to the panmictic one (panES) and the distributed GA (dGA) to the panmictic genetic algorithm (panGA). With this comparison we want to study the influence of the decentralized design in the search. For this reason we set the population size and the maximum number of evaluations in all the algorithms to the same values. The parameters of the algorithms are presented in Tables 3 and 4. In all the algorithms we use 25 individuals in the population, random selection, $(\mu+\lambda)$ replacement, and the stop criterion consists of finding a solution or reaching a maximum of 500 evaluations (in the distributed algorithms each of the five islands performs a maximum of 100 evaluations). We must remember here that the evolutionary algorithms can be executed many times during one run of the test data generator (see Section 2.3). Thus, a global number of evaluations larger than 500 might appear in the results. For the panGA we perform a previous study to determine the best configuration (see Appendix A) and we use the same configuration in dGA. The recombination operator used in these algorithms is DPX with probability 1.0. However, from the two offspring created by the operator only one is selected. In panES and dES no recombination operator is used. The mutation operator in panGA and dGA consists of adding a random value following a normal distribution with mean 0 and standard deviation 1 to each input attribute (as mentioned in Section 3.2) with probability 0.6. In the case of panES and dES the well-known Gaussian mutation (see Section 3.1) is used. The distributed algorithms use five islands connected with an asynchronous unidirectional ring topology. For the migration, one individual is selected by 2-tournament selection and sent to the neighboring island after every ten steps of the subalgorithm main loop. The individual migrated is inserted into the target subpopulation if it is better than the worst individual in that subpopulation. At the beginning of the EA search, the test data generator seeds the EAs with one individual (program input) reaching the condition associated with the current partial objective as we said in Section 2.3. The machines used for the experiments are Pentium 4 at 2.8GHz with 512MB of RAM. In the distributed algorithms (dES and dGA) each island is executed in a different machine.

We performed 30 independent runs of the test data generator for each test program. In Table 5 we present the average and the maximum corrected condition coverage percentage, the average number of evaluations (test program executions) required to get the coverage reached, and the average time of

Table 3
Parameters of the distributed algorithms dES and dGA

Parameters	dES	dGA
Population size	25	25
Selection	Uniform Random	Uniform Random
Recombination	-	DPX ($p_c = 1.0$)
Mutation	Gaussian mutation	Add $N(0,1)$ ($p_m = 0.6$)
Offspring	1 per island	1 per island
Replacement	$(\mu + \lambda)$	$(\mu + \lambda)$
Stop criterion	Objective or 500 evaluations	Objective or 500 evaluations
Islands	5	
Topology	Unidirectional Ring	
Migration type	Asynchronous	
Migration gap	10	
Migr. Selection	2-Tournament	
Migr. Rate	1	
Migr. Replacement	Local worst if incoming is best	

Table 4
Parameters of the panmictic algorithms panES and panGA

Parameters	panES	panGA
Population size	25	25
Selection	Uniform Random	Uniform Random
Recombination	-	DPX ($p_c = 1.0$)
Mutation	Gaussian mutation	Add $N(0,1)$ ($p_m = 0.6$)
Offspring	5	5
Replacement	$(\mu + \lambda)$	$(\mu + \lambda)$
Stop criterion	Objective or 500 evaluations	Objective or 500 evaluations

the whole generation process (in seconds) for dES and panES algorithms. We highlight the best results (the largest values in the case of coverage and the smallest ones in case of number of evaluations and time) in boldface.

Table 5
Results obtained with dES and panES for all the programs

	dES				panES			
Program	Avg. Cov.	Max. Cov.	Evals.	Time(s)	Avg. Cov.	Max. Cov.	Evals.	Time(s)
`triangle`	99.92	**100.00**	1898.43	**5.50**	**100.00**	**100.00**	**1207.30**	7.80
`gcd`	**100.00**	**100.00**	39.93	0.80	**100.00**	**100.00**	**15.67**	**0.53**
`calday`	**97.88**	**100.00**	**2188.17**	**7.47**	97.73	**100.00**	2586.30	23.97
`crc`	**100.00**	**100.00**	39.80	1.43	**100.00**	**100.00**	**12.83**	**0.80**
`insertion`	**100.00**	**100.00**	**10.00**	0.60	**100.00**	**100.00**	**10.00**	**0.17**
`shell`	**100.00**	**100.00**	**10.00**	0.53	**100.00**	**100.00**	**10.00**	**0.30**
`quicksort`	**94.12**	**94.12**	**10.00**	**13.57**	**94.12**	**94.12**	**10.00**	39.43
`heapsort`	**100.00**	**100.00**	**10.00**	0.47	**100.00**	**100.00**	**10.00**	**0.43**
`select`	**83.33**	**83.33**	34.17	**14.40**	**83.33**	**83.33**	**13.30**	41.03
`bessel`	**97.56**	**97.56**	481.33	**11.53**	**97.56**	**97.56**	**198.63**	32.20
`sa`	99.55	**100.00**	**1451.13**	**4330.20**	**99.77**	**100.00**	1730.43	10302.73
`netflow`	**98.17**	**98.17**	579.83	**1440.50**	**98.17**	**98.17**	**362.40**	3242.50

From the results in Table 5 we conclude that there is no difference between the two algorithms with respect to the coverage metric. In fact, in those programs where the average value differs (`triangle`, `calday`, `sa`, and `netflow`) a statistical test (details shown in Appendix B) reveals that there is no statistical difference among them. With respect to the effort, we observe a slightly higher number of evaluations in the case of dES for programs with the same coverage using both algorithms. This is somewhat unexpected, since we were aiming for new larger coverage and reduced numerical effort when using distributed ESs. For software testing, our first conclusion is that this does not hold, at least for such island model, and then only a reduction in the running time can be gained. This result is deceptive because many other works exist in which the distributed version is clearly superior to the centralized one [49,50,61], pointing out the necessity of making software testing with distributed algorithms that cooperate in a different manner, maybe by partially constructing solutions as the search progresses.

It is worth mentioning that for the `netflow` program our algorithms find several input data for which the program does not end. For this reason, if the execution of the program takes more than one minute the generator stops it in order to continue the search. This fact explains the long time observed in Table 5 (and to a lesser extent in the following results) for the `netflow` program (when the program ends normally it takes no more than one second).

We did find the expected result with respect to the execution time. The execution of dES is faster than that of panES because each subalgorithm is executed in parallel. The speed up is sublinear (up to 3.21 times faster with 5 processors), because the algorithms are not the same and there is an important core sequential part in the distributed algorithm. The startup process of the dES requires more time because of the execution of the islands in different machines. For this reason we can observe one advantage of the panES algorithm for fast programs.

A final conclusion from these results is that `insertion`, `shell`, and `heapsort` are the easiest programs of the benchmark. The test data generator is able to get full coverage only with the first 10 random inputs generated, that is, it is not necessary to use an optimization algorithm at all. These three programs are followed by two other easy-to-cover programs: `gcd` and `crc`. For these programs, even a random test data generator could get total coverage.

Now, we present in Table 6 the results obtained by the test data generator using the parallel distributed and sequential panmictic GAs as search engine.

From the results we conclude, as in the comparison between the ES-based test data generators, that there is no difference between the two algorithms in the coverage. The number of evaluations is slightly higher in the dGA, and the execution time of dGA is lower than the panGA one. That is, the way in which the search space is explored in the distributed versions of the EAs does

Table 6
Results obtained with dGA and panGA for all the programs

	dGA				panGA			
Program	Avg. Cov.	Max. Cov.	Evals.	Time(s)	Avg. Cov.	Max. Cov.	Evals.	Time(s)
`triangle`	**99.84**	**100.00**	**3004.43**	**7.53**	99.67	**100.00**	3209.47	20.30
`gcd`	**100.00**	**100.00**	445.60	**1.57**	**100.00**	**100.00**	**257.20**	2.00
`calday`	**90.91**	**90.91**	304.17	**10.43**	**90.91**	**90.91**	**75.03**	28.53
`crc`	**100.00**	**100.00**	14.83	0.83	**100.00**	**100.00**	**10.37**	**0.73**
`insertion`	**100.00**	**100.00**	**10.00**	0.47	**100.00**	**100.00**	**10.00**	**0.40**
`shell`	**100.00**	**100.00**	**10.00**	**0.37**	**100.00**	**100.00**	**10.00**	0.43
`quicksort`	**94.12**	**94.12**	**10.00**	**13.20**	**94.12**	**94.12**	**10.00**	36.43
`heapsort`	**100.00**	**100.00**	**10.00**	0.50	**100.00**	**100.00**	**10.00**	**0.33**
`select`	**83.33**	**83.33**	322.07	**14.48**	**83.33**	**83.33**	**83.20**	36.07
`bessel`	**97.56**	**97.56**	550.67	**13.57**	**97.56**	**97.56**	**533.03**	38.63
`sa`	**96.78**	**98.31**	335.33	**3865.30**	96.72	**98.31**	**176.63**	6529.63
`netflow`	96.36	97.25	937.33	**203.33**	**96.42**	**98.17**	**917.90**	240.13

not seem to improve the results with respect to the panmictic versions. This is in fact one contribution of our work since it is counterintuitive, and most people expect advantages to come from the decentralized search if we think of the overwhelming set of results on this matter published in the associated literature.

However, due to the parallel execution, the distributed versions can be suitable for time consuming programs, like `sa`. We can also parallelize the panmictic version using, for example, a master/slave approach by evaluating the individuals (program inputs) in different machines, since the evaluation operator can be performed in parallel (see the global parallelism approach in Section 3). But, in this last case the algorithm needs to be synchronized with all the machines once in each step. This synchronization can be very damaging for the execution time of the whole process, especially when the execution time of the test program depends on the input data and, thus, lots of processors could be idle for long periods of time. This idea will be investigated in future work (not here). In the distributed EA, synchronization between the machines is performed when all the subalgorithms stop and thus the number of synchronization points is very low (it is exactly the number of executions of the dEA during one run of the test data generator), reducing the overall time of the test data generation. This is why we consider it interesting to research with distributed approaches as done in this paper.

In order to confirm that `insertion`, `shell`, `heapsort`, `gcd`, and `crc` are the easiest programs in the benchmark we have used a random test data generator trying a maximum of 20000 random inputs and the results obtained are presented in Table 7.

As we expected, `crc`, `insertion`, `shell`, and `heapsort` are covered with a few random inputs. However, contrary to our expectations, `gcd` is not covered by the random testing, only 80% of the partial objectives are covered after the 20000 random inputs (and they are covered with the first three or four random inputs). The reason is that there are two equalities inside this program, which

Table 7
Results obtained with Random Testing for all the programs

	Random Test Data Generator			
Program	Avg. Cov.	Max. Cov.	Evals.	Time(s)
`triangle`	51.22	51.22	141.67	92.30
`gcd`	80.00	80.00	3.33	94.77
`calday`	95.45	95.45	74.97	95.10
`crc`	100.00	100.00	5.97	0.33
`insertion`	100.00	100.00	1.07	0.10
`shell`	100.00	100.00	1.00	0.07
`quicksort`	94.12	94.12	2.07	123.23
`heapsort`	100.00	100.00	1.00	0.10
`select`	11.11	11.11	1.00	99.87
`bessel`	70.73	70.73	158.10	97.70
`sa`	96.67	98.31	639.03	5861.73
`netflow`	95.38	96.33	9225.77	133.40

are very difficult to cover with random inputs. In addition, from the random testing results we find that the coverage obtained for the `quicksort` program is the same as that obtained with the EA-based test data generators, that is, the EA-based test data generators do not improve the results of a random search. This reveals that it is very easy to reach 94.12% coverage but very difficult to get total coverage. The reason is that there is one condition checking a memory allocation failure and another one checking a stack overflow. That is, there is an environment-dependent coverage loss that sets 94.12% as the absolute optimum for this program in the running environment. In conclusion, we do not need a metaheuristic algorithm to cover `crc`, `insertion`, `shell`, `quicksort` and `heapsort`, they can be covered by random testing. For this reason they are not used in the following sections.

4.3 Evolutionary Strategy vs Genetic Algorithm

In this section we compare the test data generators with respect to the kind of EA they use. That is, we compare the dES against the dGA and the panES against the panGA. Let us begin comparing the distributed versions in Table 8.

Table 8
Comparison between dES and dGA

	dES				dGA			
Program	Avg. Cov.	Max. Cov.	Evals.	Time(s)	Avg. Cov.	Max. Cov.	Evals.	Time(s)
`triangle`	**99.92**	**100.00**	**1898.43**	**5.50**	99.84	**100.00**	3004.43	7.53
`gcd`	**100.00**	**100.00**	**39.93**	**0.80**	**100.00**	**100.00**	445.60	1.57
`calday`	**97.88**	**100.00**	2188.17	**7.47**	90.91	90.91	**304.17**	10.43
`select`	**83.33**	**83.33**	**34.17**	**14.40**	**83.33**	**83.33**	322.07	14.48
`bessel`	**97.56**	**97.56**	**481.33**	**11.53**	**97.56**	**97.56**	550.67	13.57
`sa`	**99.55**	**100.00**	1451.13	4330.20	96.78	98.31	**335.33**	**3865.30**
`netflow`	**98.17**	**98.17**	**579.83**	1440.50	96.36	97.25	937.33	**203.33**

It can be observed that, in general terms, dES obtains a higher coverage than dGA with a lower number of evaluations (see the statistical tests in Table B.3). The two exceptions are those of the `calday` and `sa` programs, in which the number of evaluations is higher for the dES. However, this greater effort is used to increase the accuracy in the results. At this point we must indicate that the comparison of the number of evaluations is fair only when the two algorithms get the same coverage. Otherwise, we can conclude that the algorithm requiring fewer evaluations is better if the coverage obtained is higher. This is the case of `triangle` and `netflow` in Table 8. So we can state that the dES technique is more accurate than the dGA in software testing. The execution time does not show a clear trend in this case. In general, the mutation operator of the ES requires much more time than the operators of the GA. For this reason we would expect higher computation times in ES-based test data generators. However, ES explores the search space in a better way, requiring fewer evaluations than the GA and reducing the computation time. These two facts are in conflict with each other, and we can not observe a clear trend in the computation time.

Table 9
Comparison between panES and panGA

	panES				panGA			
Program	Avg. Cov.	Max. Cov.	Evals.	Time(s)	Avg. Cov.	Max. Cov.	Evals.	Time(s)
triangle	**100.00**	**100.00**	**1207.30**	**7.80**	99.67	**100.00**	3209.47	20.30
gcd	**100.00**	**100.00**	**15.67**	**0.53**	**100.00**	**100.00**	257.20	2.00
calday	**97.73**	**100.00**	2586.30	**23.97**	90.91	90.91	**75.03**	28.53
select	**83.33**	**83.33**	**13.30**	41.03	**83.33**	**83.33**	83.20	**36.07**
bessel	**97.56**	**97.56**	**198.63**	**32.20**	**97.56**	**97.56**	533.03	38.63
sa	**99.77**	**100.00**	1730.43	10302.73	96.72	98.31	**176.63**	**6529.63**
netflow	**98.17**	**98.17**	**362.40**	3242.50	96.42	**98.17**	917.90	**240.13**

In Table 9 we present the comparison between the panmictic EAs. Again, it can be seen that the panES technique is more accurate than the panGA. The first always obtains a coverage percentage higher than or equal to that obtained by the panGA. The number of evaluations required by the algorithms is smaller in panES than in panGA with the exceptions of `calday`, and `sa`; but the extra effort is justified (as in the previous results) by the higher coverage. The overall conclusion of this section is that the Evolutionary Strategy technique is more suitable for software testing than the Genetic Algorithm either with a distributed population or with a panmictic one, and this claim is assessed by statistical confidence tests (see Appendix B).

4.4 Analyzing the Distributed Approach

In this section we are going to study the influence of some parameters in the distributed algorithms not only to characterize them better but also to ensure that their previous results are not the result of naive wrong parameterizations. For the experiments we use only the dES since it obtained better results than the dGA in the previous sections. At the same time, from the benchmark programs we select `calday`, because `netflow` and `sa` programs require too much time and the rest of the programs obtained the same coverage in all the runs (so there would be no appreciable difference in the results).

4.4.1 Studying the Search Mode

Initially, we study the behavior of the algorithm when each island is searching for a different objective (*diff* search mode), as proposed in Section 2.3. In the previous experiments all the subalgorithms searched for the same partial objective (*same* search mode). But before showing the results we must discuss one small detail. The algorithms of the previous sections stop when one of the islands finds the objective, since all of them search for the same objective. Now, with the *diff* search mode we are going to change the stopping condition of the algorithm. Since all the islands are searching for their own different objectives at the same time, it seems reasonable to stop after a predefined number of evaluations has been reached in the islands, in order to solve more partial objectives in one run of the algorithm. This maximum number of evaluations is set to 500 (100 in each island, as in the previous experiments). In Table 10 we show the coverage percentage, the number of evaluations, and the time of the two different search modes. The figures presented are the average of 30 independent runs.

Table 10
Comparison of two versions of dES differing in the search mode for the `calday` program

Search Mode	Coverage	Evaluations	Time(s)
same	**97.88**	2188.17	**7.47**
diff	92.12	**693.83**	10.07

As can be observed, the best results in coverage and time are obtained when all the islands search for the same partial objective (*same* search mode). In addition, the differences in all the values are statistically significant (see Table B.5 in Appendix B). That is, the collaboration among the islands is fruitful when they are all searching for the same objective, and not when they search for different objectives.

4.4.2 Studying the Stop Criterion

Now, we are going to study the stop criterion used in dES. The algorithm used in Sections 4.2 and 4.3 stops when a program input covering the partial objective is found or a maximum number of evaluations is reached. As we said in Section 2.3, all the input data covering a partial objective not yet covered is kept and added to the coverage table, even if the objective covered is not the one being searched for. However, the algorithm stops only when the searched partial objective is found or the maximum number of evaluations is reached. We study here the alternative of stopping also when a program input is found that covers a new partial objective (not necessarily the one being searched for). In Table 11 we compare this alternative (*new* stop condition) with the previously used one (*obj* stop condition).

Table 11
Comparison of two versions of dES differing in the stop condition for the `calday` program

Stop Condition	Coverage	Evaluations	Time(s)
obj	**97.88**	**2188.17**	**7.47**
new	97.73	2640.47	9.17

We can observe in the results that the *obj* stop condition has a slight advantage over the *new* one. However, the differences are not statistically significant, so we can not definitely state that the stop condition used in the previous sections (*obj* stop condition) is better than that introduced in this section (*new* stop condition).

4.4.3 Studying the Number of Seeds

The next study we perform concerns the number of individuals used to seed the optimization algorithm. The test data generator used in the previous experiments seeds each island of dES with one individual (program input) that reaches the condition associated to the partial objective. Each island is seeded with a different individual if possible, that is, if there are enough program inputs in the coverage table reaching the condition then all the islands will be seeded with a different one. Otherwise, several islands would have the same program input. We examine in this experiment three different values for the number of seeds used in the islands. The aim of this study is to check whether the number of "good" solutions in the initial population of dES improves the accuracy of the test data generator. In Table 12 we show the results obtained when comparing three test data generators seeding dES with 1, 2, and 3 individuals, respectively.

The best coverage percentage is obtained with a lower number of seeds (which seems counterintuitive) and the difference is statistically significant with respect to the generator using only two seeds. We find a statistical difference for neither the number of evaluations nor the time. One possible reason for this unexpected behavior could be that the ES has to share its steps between the two good solutions of the population (the two seeds) and thus the offspring of both seeds reach the objective later. However, this is not observed when

Table 12
Comparison of three versions of the test data generator with a different number of seeds in the initial population of the dES for the `calday` program

Seeds	Coverage	Evaluations	Time
1	**97.88**	2188.17	**7.47**
2	94.85	**1300.90**	9.17
3	95.91	1745.30	8.53

three seeds are used. In this case some seeds are definitely repeated in the population (because only ten solutions are stored in the coverage table for each partial objective) and this can benefit the evolution of the redundant individuals. However, more experiments are needed to confirm (or refute) this hypothesis.

4.4.4 Studying the Migration Gap

Finally, we are going to study the influence of the migration gap of the islands in the results. The migration gap is the number of steps between two consecutive migrations and is a measure of the coupling between the islands in a distributed EA. In the previous experiments the migration gap was set to 10. In this section we try four more values: 30, 50, 70, and 90. Since the number of maximum steps of a subalgorithm is 100, a migration gap of 90 means almost no communication between the islands. We want to test with this experiment whether the collaboration of the islands profits the search or not. In Table 13 we show the results.

Table 13
Comparison of five versions of the test data generator with a different migration gap in dES for the `calday` program

Migration Gap	Coverage	Evaluations	Time
10	97.88	2188.17	7.47
30	98.18	**1861.13**	7.27
50	98.33	2085.97	7.20
70	98.94	2073.27	7.13
90	**99.09**	2005.50	**6.23**

We can observe that the results improve when the migration gap increases. This is not surprising because as we saw in Section 4.2, the distributed approach is not better than the panmictic one. However, the values of Table 13 are not statistically significant.

4.5 Previous Results

The task of comparing our results against previous work is a difficult one. First, we need to find papers tackling the same test programs. There is one test program which is very popular in the domain of software testing: the triangle classifier. However, there are several different implementations in the literature and the source code is usually not provided. In [21] the source code of the triangle classifier is published. In this work, we use that implementation for comparing performance. We have two other test programs in common with [21]: the computation of the greatest common denominator and the insertion sort. However, we use different implementations for these algorithms. In order to facilitate future comparisons we have indicated in Table 2 how to get the source code of the test programs (available at URL `http://tracer.lcc.uma.es/problems/testing/` with the exception of `netflow`).

A second obstacle when comparing different techniques is that of the metrics. We use the corrected condition coverage to measure the quality of the solutions. In [34,40,41] the authors report only on the branch coverage. On the other hand, the coverage metric used in [21] is obtained by using a proprietary software: `DeepCover`. We can not directly compare all these results in a quantitative manner because all the papers use different metrics. Another obstacle which may affect the results is the number of independent runs. A low number of independent runs in stochastic algorithms is not enough to obtain a clear idea about the behavior of the technique used. This is not our case, but some papers perform too few independent runs.

In spite of all the previous considerations, we include in Table 14 the best average coverage results reported for the triangle classifier algorithm in [21,34,40,41] and the necessary number of evaluations (program tests). We show in the same table the results of our panmictic ES, the best with regard to the coverage percentage.

Table 14
Previous results of coverage and number of evaluations for `triangle`

`triangle`	Ref. [21]	Ref. [34]	Ref. [40]	Ref. [41]	panES (here)
Coverage (%)	94.29[b]	**100.00**[a]	**100.00**[a]	**100.00**[a]	**100.00**[c]
Evaluations	$\approx$ 8000	18000	**608**	3439	1207.30

[a]Branch coverage.
[b]`DeepCover` coverage.
[c]Corrected condition coverage.

If we focus on the coverage of Table 14, our panES has the best coverage as well as the algorithms in [34,40,41]. Comparing the number of evaluations, our work ranks in second position with respect to the best (lowest) value computed in the work of Sagarna and Lozano [40]. However, we must remember that the condition coverage (used here in our work) is a more difficult test adequacy criterion than the branch coverage used by the authors in [40] (see Section 2), and this slight increment in the number of evaluations of our algorithm is justified since it achieves a result of higher quality.

We also have a program in common with the work of Wegener et al. [60]: `netflow`. The comparison is shown in Table 15. Analyzing the code we found that our 98.17% of corrected coverage corresponds to the 99% of branch coverage they report (which according to their work is the very maximum coverage). The conclusion is that we obtain the same coverage with a hundredth of the number of evaluations, despite the fact that they use a more sophisticated fitness function.

Table 15
Previous results of coverage and number of evaluations for `netflow`

`netflow`	Ref. [60]	panES (here)
Coverage (%)	99[a]	98.17[b]
Evaluations	40703	362.40

[a]Branch coverage.
[b]Corrected condition coverage.

5 Conclusions

In this article we have analyzed the application of sequential and parallel decentralized EAs to the automatic test data generation problem. In particular, we have compared a distributed ES and a distributed GA to their panmictic versions using a benchmark of twelve test programs implementing some fundamental algorithms in computer science.

The results show that the decentralized versions have no statistically significant advantage over the panmictic versions, neither in terms of the coverage nor in effort. This is an unexpected observation since much research exists reporting a higher degree of accuracy for the decentralized approach. The conclusion is that the decentralized algorithms should maybe focus on cooperating by constructing the solutions and that other information should perhaps be migrated (such as subalgorithm parameters or average entropy) that influences in the way the search is performed on each island.

On the other hand, comparing the two EA techniques, either with a distributed or panmictic population, we can state that our proposed ES outperforms the results of the GA. This opens a promising line of future research concerning the application of ES to evolutionary testing. Furthermore, the number of parameters to be tuned in the ES are fewer and for this reason we believe is more suitable for automatic tools that must be used by people with no knowledge about evolutionary algorithms.

In a second stage we have studied the influence of several parameters of the dES-based test data generator such as the search mode, the stop condition, the number of seeds used in the dES islands and the migration gap. The results state that, by searching for the same partial objective in all the islands, we can outperform the results with respect to the version that searches for different partial objectives in the islands. On the other hand, the stop condition that involves stopping when any new partial objective is covered does not have a clear advantage over the one in which only the coverage of the searched partial objective is used to stop the algorithm. Analyzing the number of seeds used in the initial population of dES, we discovered that the best results are obtained with one single seed. Finally, a high migration gap seems to benefit the search and this confirms that the distributed approach is not good for this problem.

As future work, we plan to study the combination of the ES with other techniques and propose some new methods based on static analysis that can be applied as a local search operator inside the evolutionary algorithm scheme in order to hopefully increase the coverage (more accuracy) and decrease the computational effort (more efficiency) required in the test case generation. We also want to extend the input data generation to other non-numeric data types such as strings, structures, arrays, and so on. On the other hand, an interesting idea is to study the ways in which automatic software testing can be applied to reactive software.

Acknowledgements

We thank to Joachim Wegener for easing the access to the source code of the `netflow` program. The authors acknowledge partial funding by the Ministry of Education and Science and FEDER under contract TIN2005-08818-C04-01 (the OPLINK project). Francisco Chicano is supported by a grant (BOJA 68/2003) from the Junta de Andalucía (Spain).

A The Parameters of the GA

In order to make a fair comparison between the two kinds of EAs we need to adjust the parameters of the algorithms and compare the results obtained with the best configuration of them. For this reason we try different parameters in this appendix for the mutation and the crossover of the GA. We do not tune the ES because the parameters of the mutation operator are well-established in the literature [43,56].

We are going to modify the standard deviation of the normal mutation, the mutation probability, the recombination operator, and the recombination probability. We do not use the `sa` program for the experiments because it is very time consuming. In all the cases we use 25 individuals in the population, random selection, $(\mu + \lambda)$ replacement, and the stop criterion consists of finding a solution or reaching a maximum of 500 evaluations. For the first experiment we use uniform crossover with probability 1.0 and normal mutation with probability 0.2. The mean of the mutation is 0 and we try three different values for the standard deviation: 1, 10, and 100. The results are in Table A.1. We do not show the results of `insertion`, `shell`, `quicksort`, and `heapsort` because they always get the same results and are not meaningful for the study.

Table A.1
Results obtained when changing the standard deviation of the mutation

	$\sigma = 1$		$\sigma = 10$		$\sigma = 100$	
Program	Avg. Cov.	Evals.	Avg. Cov.	Evals.	Avg. Cov.	Evals.
`triangle`	**99.43**	3961.27	98.86	**3175.30**	97.15	4718.13
`gcd`	**100.00**	**190.93**	**100.00**	253.73	**100.00**	1213.80
`calday`	90.91	224.30	90.91	**114.30**	**91.97**	1548.00
`crc`	**100.00**	**10.30**	**100.00**	10.80	**100.00**	10.60
`select`	**83.33**	196.37	**83.33**	**171.70**	**83.33**	517.93
`bessel`	**97.56**	**264.13**	**97.56**	293.27	**97.56**	1038.87
`netflow`	96.33	525.47	96.12	**454.07**	**96.36**	706.10

From the table we can see that the standard deviation has a different influence in each program. This is not surprising: this means that each problem has its own best parameterization. Here, a low standard deviation seems to be good for the coverage of `triangle` but not so good for the coverage of `calday` and `netflow`. However, most of the differences are not statistically significant. Only in the case of `triangle` and `calday` we can find significant differences and they are contradictory. On the other hand, the number of evaluations is higher with statistical significance in `gcd`, `calday`, and `bessel` for standard deviation 100. In this case we decide to keep a low standard deviation, that is, the perturbation in the solution is not very large. For this reason we set the standard deviation of the mutation to 1 for the following experiments. In the next experiment we analyze five different values for the mutation probability: 0.2, 0.4, 0.6, 0.8, and 1.0. The results are in Table A.2.

We can not observe a clear influence of the mutation probability in the coverage. In fact, the statistical tests show that the differences are not significant. However, with respect to the number of evaluations we observe a decrease in several programs when the probability is higher. Again, the differences are not

Table A.2
Results obtained when changing the mutation probability

	$p_m = 0.2$		$p_m = 0.4$		$p_m = 0.6$		$p_m = 0.8$		$p_m = 1.0$	
Program	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.
`triangle`	99.43	3961.27	**99.76**	3057.07	99.19	3175.23	99.51	4033.43	99.51	**2327.27**
`gcd`	**100.00**	190.93	**100.00**	193.33	**100.00**	153.43	**100.00**	165.60	**100.00**	**119.40**
`calday`	**90.91**	224.30	**90.91**	109.73	**90.91**	82.80	**90.91**	84.73	**90.91**	**67.30**
`crc`	**100.00**	**10.30**	**100.00**	10.83	**100.00**	10.37	**100.00**	10.57	**100.00**	10.50
`select`	**83.33**	196.37	**83.33**	173.83	**83.33**	**112.70**	**83.33**	124.33	**83.33**	117.00
`bessel`	**97.56**	264.13	**97.56**	182.97	**97.56**	198.73	**97.56**	188.80	**97.56**	**135.50**
`netflow`	96.33	525.47	96.33	710.80	**96.39**	789.53	96.33	**473.40**	96.36	692.53

significant (except in the case of `bessel` for probabilities 0.2 and 1.0) so we can not conclude that one probability value is better than another one. We set the probability of the mutation to an intermediate value: 0.6. Next, we are going to try three different crossover operators: uniform crossover, single point crossover, and double point crossover. The results are in Table A.3.

Table A.3
Results obtained when changing the crossover operator

	Uniform		Single Point		Double Point	
Program	Avg. Cov.	Evals.	Avg. Cov.	Evals.	Avg. Cov.	Evals.
`triangle`	99.19	**3175.23**	99.51	4088.60	**99.67**	3209.47
`gcd`	**100.00**	**153.43**	**100.00**	317.77	**100.00**	257.20
`calday`	**90.91**	82.80	**90.91**	255.07	**90.91**	**75.03**
`crc`	**100.00**	**10.37**	**100.00**	10.80	**100.00**	**10.37**
`select`	**83.33**	112.70	**83.33**	110.43	**83.33**	**83.20**
`bessel`	**97.56**	198.73	**97.56**	**185.77**	**97.56**	533.03
`netflow`	96.39	789.53	96.33	**626.27**	**96.42**	917.90

We observe a slight advantage (not significant) of the double point crossover with respect to the coverage. For this reason we select this operator. In general, however, the number of evaluations is also higher with this operator. The last experiment in this appendix is used to select the crossover probability. We try five values for the probability: 0.2, 0.4, 0.6, 0.8, and 1.0. We show the results in Table A.4.

Table A.4
Results obtained when changing the crossover probability

	$p_c = 0.2$		$p_c = 0.4$		$p_c = 0.6$		$p_c = 0.8$		$p_c = 1.0$	
Program	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.	Cov.	Evals.
`triangle`	98.13	4814.97	97.97	**3208.60**	99.02	3817.57	99.51	3310.90	**99.67**	3209.47
`gcd`	**100.00**	**117.53**	**100.00**	131.03	**100.00**	145.20	**100.00**	210.17	**100.00**	257.20
`calday`	**90.91**	75.37	**90.91**	92.97	**90.91**	**58.83**	**90.91**	75.00	**90.91**	75.03
`crc`	**100.00**	11.03	**100.00**	11.30	**100.00**	11.13	**100.00**	**10.00**	**100.00**	10.37
`select`	**83.33**	92.07	**83.33**	99.50	**83.33**	93.50	**83.33**	107.77	**83.33**	**83.20**
`bessel`	**97.56**	**127.10**	**97.56**	170.10	**97.56**	193.17	**97.56**	264.90	**97.56**	533.03
`netflow`	96.36	1277.07	96.36	1236.60	96.36	1060.63	96.33	**902.67**	**96.42**	917.90

In general, we observe a better coverage and worse efficiency with higher probabilities. The few differences that are statistically significant support this observation. For this reason we select the higher probability 1.0 for the double point crossover operator. The final configuration for the GA is a double point crossover with probability 1.0 and a normal mutation with mean 0, standard deviation 1, and probability 0.6. This is the configuration of the GA-based generators in the experiments of Section 4.

B Statistical Validation of the Results

In this appendix we include the statistical tests performed for the comparisons of this article (except those of Appendix A). This is a very important practice that researchers in metaheuristics and in non-deterministic algorithms in general should include in their work. Nowadays, authors not doing statistical tests often report "clear" advantages for their proposals based on rather small negligible numerical improvements. Although it is an intensive and time consuming task, work including this information is supposed to improve the overall research quality in literature.

In each case the procedure for generating the statistical information presented in the tables is the following. First a Kolmogorov-Smirnov test is performed in order to check whether the variables are normal or not. If they are, an ANOVA I test is performed, otherwise we perform a Kruskal-Wallis test. After that, we do a multiple comparison test whose results we present in the following tables. We highlight with boldface the values associated with a significant difference. This occurs when zero is not included between the lower and upper bounds of the confidence interval (see columns *Lower* and *Upper*). We do not show the results of the statistical tests for `insertion`, `shell`, `quicksort`, and `heapsort` because they get the same results in coverage and number of evaluations.

Table B.1
Statistical test results for the comparison between dES and panES

	Coverage			Evaluations			Time		
Program	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
`triangle`	-2.96	-1.00	0.96	**37.19**	**691.13**	**1345.07**	**-20.50**	**-11.73**	**-2.97**
`gcd`	0.00	0.00	0.00	**19.23**	**28.00**	**36.77**	**0.78**	**8.00**	**15.22**
`calday`	-5.79	1.33	8.46	-10.90	-2.07	6.77	**-20.30**	**-16.50**	**-12.70**
`crc`	0.00	0.00	0.00	-1.89	3.60	9.09	-0.12	7.70	15.52
`select`	0.00	0.00	0.00	**17.57**	**26.33**	**35.10**	**-38.53**	**-30.00**	**-21.47**
`bessel`	0.00	0.00	0.00	**2.04**	**10.87**	**19.69**	**-38.76**	**-30.00**	**-21.24**
`sa`	-10.12	-4.00	2.12	-1136.57	-279.30	577.97	**-9284.62**	**-5972.53**	**-2660.45**
`netflow`	0.00	0.00	0.00	**95.06**	**217.43**	**339.80**	**-2020**	**-1800**	**-1580**

Table B.2
Statistical test results for the comparison between dGA and panGA

	Coverage			Evaluations			Time		
Program	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
`triangle`	-3.17	1.07	5.30	-1487.02	-205.03	1076.96	**-17.37**	**-12.77**	**-8.16**
`gcd`	0.00	0.00	0.00	**94.54**	**188.40**	**282.26**	-11.87	-3.57	4.74
`calday`	0.00	0.00	0.00	**186.28**	**229.13**	**271.98**	**-38.52**	**-30.00**	**-21.48**
`crc`	0.00	0.00	0.00	-2.27	1.07	4.40	-7.01	0.63	8.27
`select`	0.00	0.00	0.00	**170.96**	**238.87**	**306.78**	**-38.12**	**-29.50**	**-20.88**
`bessel`	0.00	0.00	0.00	-165.54	17.63	200.80	**-38.74**	**-30.00**	**-21.26**
`sa`	-3.23	1.00	5.23	-3.17	1.07	5.30	**-3076.36**	**-2664.33**	**-2252.30**
`netflow`	-4.37	-1.03	2.30	-416.70	19.43	455.57	**-23.83**	**-15.00**	**-6.17**

Table B.3
Statistical test results for the comparison between dES and dGA

	Coverage			Evaluations			Time		
Program	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
`triangle`	-2.34	1.00	4.34	-2308.16	-1106.00	96.16	-14.31	-5.57	3.18
`gcd`	0.00	0.00	0.00	**-463.64**	**-405.67**	**-347.69**	**-26.35**	**-18.80**	**-11.25**
`calday`	**16.42**	**24.00**	**31.58**	**9.33**	**18.17**	**27.00**	**-24.39**	**-15.90**	**-7.41**
`crc`	0.00	0.00	0.00	-0.95	4.27	9.49	-0.19	7.27	14.73
`select`	0.00	0.00	0.00	**-350.46**	**-287.90**	**-225.35**	-10.20	-2.24	5.73
`bessel`	0.00	0.00	0.00	**-24.20**	**-15.37**	**-6.54**	**-32.42**	**-23.73**	**-15.05**
`sa`	**21.03**	**29.20**	**37.37**	**16.72**	**25.13**	**33.55**	-594.47	464.90	1524.27
`netflow`	**22.28**	**30.00**	**37.72**	**-613.60**	**-357.50**	**-101.40**	**19.16**	**28.00**	**36.84**

Table B.4
Statistical test results for the comparison between panES and panGA

	Coverage			Evaluations			Time		
Program	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
`triangle`	-0.34	3.00	6.34	**-2793.34**	**-2002.17**	**-1210.99**	**-16.84**	**-12.50**	**-8.16**
`gcd`	0.00	0.00	0.00	**-38.49**	**-29.70**	**-20.91**	**-27.85**	**-19.53**	**-11.22**
`calday`	**17.27**	**25.00**	**32.73**	**1709.70**	**2511.27**	**3312.84**	-12.02	-3.40	5.22
`crc`	0.00	0.00	0.00	-1.79	2.03	5.86	-7.70	0.20	8.10
`select`	0.00	0.00	0.00	**-31.45**	**-22.67**	**-13.88**	**21.39**	**30.00**	**38.61**
`bessel`	0.00	0.00	0.00	**-27.70**	**-18.87**	**-10.03**	**-35.79**	**-27.00**	**-18.21**
`sa`	**21.75**	**29.73**	**37.72**	**18.82**	**27.20**	**35.58**	**608.07**	**3773.10**	**6938.13**
`netflow`	**21.29**	**29.00**	**36.71**	**-929.13**	**-555.50**	**-181.87**	**2794.27**	**3002.37**	**3210.47**

Table B.5
Statistical test results for the different search modes in dES applied to the `calday` program

	Coverage			Evaluations			Time		
Search Mode	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
same vs *diff*	**11.99**	**19.73**	**27.47**	**6.76**	**15.60**	**24.44**	**-22.59**	**-14.00**	**-5.41**

Table B.6
Statistical test results for the different stop conditions in dES applied to the `calday` program

	Coverage			Evaluations			Time		
Stop Condition	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
obj vs *new*	-5.38	1.87	9.12	-11.70	-2.87	5.97	-3.84	-1.70	0.44

Table B.7
Statistical test results for the different number of seed individuals in dES applied to the `calday` program

	Coverage			Evaluations			Time		
Seeds	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
1 vs 2	**1.96**	**15.83**	**29.70**	-1.71	14.43	30.58	-30.04	-14.43	1.18
1 vs 3	-3.70	10.17	24.04	-6.93	9.22	25.36	-23.23	-7.62	7.99
2 vs 3	-19.54	-5.67	8.20	-21.36	-5.22	10.93	-8.79	6.82	22.43

Table B.8
Statistical test results for the migration gap in dES applied to the `calday` program

	Coverage			Evaluations			Time		
Seeds	Lower	Estim.	Upper	Lower	Estim.	Upper	Lower	Estim.	Upper
10 vs 30	-22.62	0.43	23.49	-26.96	3.63	34.23	-31.68	-0.38	30.92
10 vs 50	-24.91	-1.85	21.21	-35.66	-5.07	25.53	-31.62	-0.32	30.98
10 vs 70	-29.04	-5.98	17.07	-35.85	-5.25	25.35	-30.07	1.23	32.53
10 vs 90	-32.99	-9.93	13.12	-39.91	-9.32	21.28	-21.00	10.30	41.60
30 vs 50	-25.34	-2.28	20.77	-39.30	-8.70	21.90	-31.23	0.07	31.37
30 vs 70	-29.47	-6.42	16.64	-39.48	-8.88	21.71	-29.68	1.62	32.92
30 vs 90	-33.42	-10.37	12.69	-43.55	-12.95	17.65	-20.62	10.68	41.98
50 vs 70	-27.19	-4.13	18.92	-30.78	-0.18	30.41	-29.75	1.55	32.85
50 vs 90	-31.14	-8.08	14.97	-34.85	-4.25	26.35	-20.68	10.62	41.92
70 vs 90	-27.01	-3.95	19.11	-34.67	-4.07	26.53	-22.23	9.01	40.37

References

[1] E. W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.

[2] E. W. Dijkstra, C. S. Scholten, Predicate Calculus and Program Semantics, Springer-Verlag, New York, 1990.

[3] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.

[4] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, The MIT Press, 2000.

[5] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: Logic of Programs, Workshop, Springer-Verlag, London, UK, 1982, pp. 52–71.

[6] E. M. Clarke, E. A. Emerson, A. P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Trans. Program. Lang. Syst. 8 (2) (1986) 244–263.

[7] G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 1–17.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, H. Zheng, Bandera: extracting finite-state models from java source code, in: ICSE '00: Proceedings of the 22nd international conference on Software engineering, ACM Press, New York, NY, USA, 2000, pp. 439–448.

[9] D. Y. W. Park, U. Stern, J. U. Skakkebaek, D. L. Dill, Java model checking, in: Proceedings of Automated Software Engineering conference, 2000.

[10] S. Bensalem, V. Ganesh, Y. Lakhnech, C. M. noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, A. Tiwari, An overview of SAL, in: C. M. Holloway (Ed.), LFM 2000: Fifth NASA Langley Formal Methods Workshop, NASA Langley Research Center, Hampton, VA, 2000, pp. 187–196.

[11] A. Groce, W. Visser, Heuristics for model checking java programs 6 (4) (2004) 260–276.

[12] A. Lluch-Lafuente, S. Leue, S. Edelkamp, Partial order reduction in directed model checking, in: 9th International SPIN Workshop on Model Checking Software, Springer, Grenoble, 2002.

[13] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, D. L. Dill, Symbolic model checking for sequential circuit verification, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13 (4).

[14] A. Lluch-Lafuente, Symmetry reduction and heuristic search for error detection in model checking, in: Workshop on Model Checking and Artificial Intelligence, 2003.

[15] D. Alberts, The economics of software quality assurance, in: Proceedings of the 1976 National Computer Conference, Vol. 45, AFIPS Press, 1976, pp. 433–442.

[16] G. Myers, The Art of Software Testing, John Wiley and Sons, New York, 1979.

[17] R. D. Millo, W. McCracken, R. Martin, J. Passafiume, Software Testing and Evaluation, Benjamin/Cummings, Menlo Park, California, 1987.

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

[19] W. Miller, D. L. Spooner, Automatic generation of floating-point test data, IEEE Trans. Software Eng. 2 (3) (1976) 223–226.

[20] P. McMinn, Search-based software test data generation: a survey, Software Testing, Verification and Reliability 14 (2) (2004) 105–156.

[21] C. C. Michael, G. McGraw, M. A. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27 (12) (2001) 1085–1110.

[22] P. McMinn, M. Holcombe, The state problem for evolutionary testing, in: E. C. P. et al. (Ed.), Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2724, Springer-Verlag, Chicano, Illinois, USA, 2003, pp. 2488–2498.

[23] N. Mansour, M. Salame, Data generation for path testing, Software Quality Journal 12 (2) (2004) 121–136.

[24] K. Sen, D. Marinov, G. Agha, CUTE: a concolic unit testing engine for C, in: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, ACM Press, New York, NY, USA, 2005, pp. 263–272.

[25] P. Godefroid, N. Klarlund, K. Sen, DART: directed automated random testing, in: PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM Press, New York, NY, USA, 2005, pp. 213–223.

[26] O. Buehler, J. Wegener, Evolutionary functional testing of an automated parking system, in: Proceedings of the International Conference on Computer, Communication and Control Technologies, Orlando, Florida, 2003.

[27] O. Buehler, J. Wegener, Evolutionary functional testing of a vehicle brake assistant system, in: Proceedings of the 6th Metaheuristic International Conference, Vienna, Austria, 2005, pp. 157–162.

[28] B. Korel, A. M. Al-Yami, Assertion-oriented automated test data generation, in: Proceedings of the 18th International Conference on Software Engineering, IEEE Computer Society Press, Berlin, Germany, 1996, pp. 71–80.

[29] H. Sthamer, J. Wegener, A. Baresel, Using evolutionary testing to improve efficiency and quality in software testing, in: Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review, Melbourne, Australia, 2002.

[30] J. Wegener, H. Sthamer, B. F. Jones, D. E. Eyres, Testing real-time systems using genetic algorithms, Software Quality Journal 6 (1997) 127–135.

[31] D. Bird, C. Munoz, Automatic generation of random self-checking test cases, IBM Systems Journal 22 (3) (1983) 229–245.

[32] J. Offutt, An integrated automatic test data generation system, Journal of Systems Integration 1 (3) (1991) 391–409.

[33] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (8) (1990) 870–879.

[34] B. Jones, H. Sthamer, D. Eyres, Automatic structural testing using genetic algorithms, Software Engineering Journal 11 (5) (1996) 299–306.

[35] T. Mantere, J. T. Alander, Evolutionary software engineering, a review, Applied Soft Computing 5 (3) (2005) 315–331.

[36] D. Ostrowski, R. Reynolds, Knowledge-based software testing agent using evolutionary learning with cultural algorithms, in: Proceedings of the Congress on Evolutionary Computation, Vol. 3, 1999, pp. 1657–1663.

[37] N. Tracey, A search-based automated test-data generation framework for safety-critical software, Ph.D. thesis, University of York (2000).

[38] N. Tracey, J. Clark, K. Mander, J. McDermid, An automated framework for structural test-data generation, in: Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA, 1998, pp. 285–288.

[39] E. Díaz, J. Tuya, R. Blanco, Automated software testing using a metaheuristic technique based on tabu search, in: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada, 2003, pp. 310–313.

[40] R. Sagarna, J. Lozano, Variable search space for software testing, in: Proceedings of the International Conference on Neural Networks and Signal Processing, Vol. 1, IEEE Press, 2003, pp. 575–578.

[41] R. Sagarna, J. A. Lozano, Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms, European Journal of Operational Research 169 (2) (2006) 392–412, (available online).

[42] E. Alba, J. F. Chicano, Software testing with evolutionary strategies, in: Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques, Vol. 3943 of LNCS, Heraklion, Crete, Greece, 2005, pp. 50–65.

[43] T. Bäck, Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms, Oxford University Press, New York, 1996.

[44] E. Alba, M. Tomassini, Parallelism and evolutionary algorithms, IEEE Transactions on Evolutionary Computation 6 (5) (2002) 443–462.

[45] D. Levine, Users guide to the PGAPack parallel genetic algorithm library, Tech. Rep. ANL-95/18, Argonne National Laboratory, Mathematics and Computer Science Division (January 31st, 1995).

[46] T. C. Belding, The distributed genetic algorithm revisited, in: L. J. Eshelman (Ed.), Proceedings of the Sixth International Conference on Genetic Algorithms, Morgan Kaufmann, 1995, pp. 114–121.

[47] R. Tanese, Distributed genetic algorithms, in: J. D. Schaffer (Ed.), Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann, 1989, pp. 434–439.

[48] S. Baluja, Structure and performance of fine-grain parallelism in genetic search, in: S. Forrest (Ed.), Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993, pp. 155–162.

[49] E. Alba, J. M. Troya, Gaining new fields of application for OOP: the parallel evolutionary algorithm case, Journal of Object Oriented Programming.

[50] V. S. Gordon, D. Whitley, Serial and parallel genetic algorithms as function optimizers, in: S. Forrest (Ed.), Proceedings of the Fifth International Conference on Genetic Algorithms, Morgan Kaufmann, 1993, pp. 177–183.

[51] J. H. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, Michigan, 1975.

[52] I. Rechenberg, Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Fromman-Holzboog Verlag, Stuttgart, 1973.

[53] L. J. Fogel, A. J. Owens, M. J. Walsh, Artificial Intelligence through Simulated Evolution, John Wiley & Sons, New York, 1966.

[54] J. R. Koza, Genetic Programming, The MIT Press, Cambridge, Massachusetts, 1992.

[55] T. Bäck, D. B. Fogel, Z. Michalewicz (Eds.), Evolutionary Computation 1. Basic Algorithms and Operators, Vol. 1, IOP Publishing Lt, 2000.

[56] G. Rudolph, Evolutionary Computation 1. Basic Algorithms and Operators, Vol. 1, IOP Publishing Lt, 2000, Ch. 9, Evolution Strategies, pp. 81–88.

[57] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, K. Karapoulios, Application of genetic algorithms to software testing, in: Proceedings of the 5th International Conference on Software Engineering and its Applications, Toulouse, France, 1992, pp. 625–636.

[58] H.-H. Sthamer, The automatic generation of software test data using genetic algorithms, Ph.D. thesis, University of Glamorgan (November 1995).

[59] E. Alba, J. M. Troya, A survey of parallel distributed genetic algorithms, Complexity 4 (1999) 31–52.

[60] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test environment for automatic structural testing, Information and Software Technology 43 (14) (2001) 841–854.

[61] E. Alba (Ed.), Parallel Metaheuristics. A New Class of Algorithms, John Wiley & Sons, 2005.