Dealing with Inheritance in OO Evolutionary Testing

Javier Ferrer
University of Málaga
Spain
ferrer@lcc.uma.es

Francisco Chicano
University of Málaga
Spain
chicano@lcc.uma.es

Enrique Alba
University of Málaga
Spain
eat@lcc.uma.es

ABSTRACT

Most of the software developed in the world follows the object-oriented (OO) paradigm. However, the existing work on evolutionary testing is mainly targeted to procedural languages. All this work can be used with small changes on OO programs, but object orientation introduces new features that are not present in procedural languages. Some important issues are polymorphism and inheritance. In this paper we want to make a contribution to the inheritance field by proposing some approaches that use the information of the class hierarchy for helping test case generators to better guide the search. To the best of our knowledge, no work exists using this information to propose test cases. In this work we define a branch distance for logical expressions containing the `instanceof` operator in Java programs. In addition to the distance measure, we propose two mutation operators based on the distance. We study the behaviour of the mutation operators on a benchmark set composed of nine OO programs. The results show that the information collected from the class hierarchy helps in the search for test cases.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic methods*; G.1.6 [**Numerical Analysis**]: Optimization—*Global optimization*

General Terms

Algorithms, Experimentation

Keywords

Software Testing , Object-Oriented , Evolutionary Algorithm, OO Evolutionary Testing, `instanceof` , Search Based Software Engineering

1. INTRODUCTION

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE). From the first works [11] to nowadays many approaches have been proposed for solving the automatic test case generation problem. This great effort in building computer aided software testing tools is motivated by the cost and importance of the testing phase in the software development cycle. It is estimated that half the time spent on software project development, and more than half its cost, is devoted to testing the product [4]. This explains why the Software Industry and Academia are interested in automatic tools for testing.

Evolutionary algorithms (EAs) have been the most popular search algorithms for generating test cases [8]. In fact, the term *evolutionary testing* is used to refer to this approach. In the paradigm of *structural testing* a lot of research has been performed using EA and in particular, different elements of the structure of a program have been studied in detail. Some examples are the presence of flags in conditions [3], the coverage of loops [5], the existence of internal states [17] and the presence of possible exceptions [13].

In the field of evolutionary testing, the area of test case generation for object-oriented (OO) programs is receiving increased attention [12]. Most of the work found in the literature related to automatic software testing focuses on procedural languages and programs. However, most of the software developed nowadays is object-oriented. OO languages are considered an evolution of procedural languages and they allow developers to design and implement really large software applications more easily than when using procedural languages, thanks to their ability for abstracting and modularizing software components.

Most of the ideas used for testing procedural programs can be used with no change in the object-oriented software. However, we can find new features in OO languages that do not exist in procedural ones. Some examples of these features are inheritance, polymorphism, overloading, generics, and so on. No doubt these new features solve some common errors found in procedural languages but they also trigger new kinds of errors. Thus, when dealing with OO programs, automatic software testing techniques must include new ideas that are not present in procedural testing.

Despite the importance of OO software in industry not much work can be found in the literature on testing OO software with EAs. One early example is the work by Tonella [12], which used genetic algorithms. Later, Liu et al. [7] proposed the use of ant colony optimization. The work by Wappler and Wegener [14, 15] using hybrid algorithms and strongly-

typed genetic programming focuses on container classes. Recently, the work by Arcuri and Yao [1] optimizes test cases for covering all the branches at the same time and compares different search strategies.

None of the previous work explicitly deals with inheritance for generating test cases. The objects used as parameters are generated by calling the constructor and some methods of one class in order to reach a given state of the object. However, no attention is paid to the position of the class in the hierarchy, thus we wish to address this issue in this work. We propose new approaches to take into account the position in the class hierarchy of the classes used in the parameters of methods being tested. In particular, this work focuses on the Java `instanceof` operator, which is related to inheritance. The main and motivating question addressed here is the following: what is the guidance for an automatic test case generator if there is a method containing the code in Fig. 1 and it needs to make the branch condition true? We can use an objective function for this branch that takes value 1 if the condition is true and 0 otherwise. However, this objective function gives no clue on how near a test case is from making the condition true and an algorithm using this function can behaves like a blind search. To the best of our knowledge no work exists dealing with such situation.

```
void function (Collection c)
{
     if (c instanceof Set)
     {
     ...
     }
}
```

Figure 1: Instanceof expression in a sentence.

The specific contributions of this work are a new branch distance defined for the `instanceof` operator, and two mutation operators taking into account the class hierarchy. The proposal made in this paper allows the algorithms to better guide the search for test cases in the presence of the `instanceof` operator. This operator appears in 2,700 of the 13,000 classes of the JDK 1.6 class hierarchy and more than 850 classes include this operator in between 1% and 12% of their source code lines. This means that this operator is used in real software and any contribution that facilitates the testing process when it is present will have an important impact on the OO software testing field.

The rest of the paper is organized as follows. We describe our test data generator for Java programs in the next section. Section 3 presents our first contribution: a formal definition of the distance measure proposed for the `instanceof` operator. Then, in Section 4 we describe the EA used in this work and the proposed mutation operators (second contribution). Section 5 describes the experiments performed and discusses the results obtained. Finally, in Section 6 some conclusions are outlined.

2. THE TEST CASE GENERATOR

In this work we use *branch coverage* as adequacy criterion. This criterion is used in most of the related papers in the literature. Our test case generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Then, each partial objective can be treated as a separate optimization problem in which the function to be minimized is a distance between the current test case and one satisfying the partial objective. In order to solve such minimization problem EAs are used. The main loop of the test data generator is shown in Fig. 2.

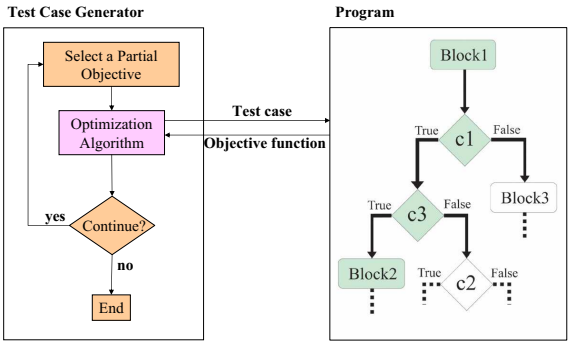

Figure 2: The test case generation process.

In a loop, the test case generator selects a partial objective (a branch) and uses the optimization algorithm to search for test cases exercising that branch. When a test case covers a branch, the test case is stored in a set associated to that branch. The structure composed of the sets associated to all the branches is called *coverage table*. After the optimization algorithm stops, the main loop starts again and the test case generator selects a different branch. This scheme is repeated until total branch 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 returns the sets of test cases associated to all the branches. In the rest of this section we describe two important issues relating to the test case generator: the objective function to minimize and the class instantiation.

2.1 Objective Function

Following on from the discussion in the previous section, we have to solve several minimization problems: one for each branch. Now we need to define an objective function (for each branch) to be minimized. This function will be used for evaluating each test case, and its definition depends on the desired branch and whether the program flow reaches the branching condition associated to the target branch or not. If the condition is reached we can define the objective function on the basis of the logical expression of the branching condition and the values of the program variables when the condition is reached. The resulting expression is called *branch distance* and can be defined recursively on the structure of the logical expression. That is, for an expression composed of other expressions joined by logical operators the branch distance is computed as an aggregation of the branch distance applied to the component logical expressions. For the Java logical operators `&` and `|` we define the branch distance as[1]:

$$bd(a\&b) = bd(a) + bd(b) \quad (1)$$

$$bd(a|b) = \min(bd(a), bd(b)) \quad (2)$$

where a and b are logical expressions.

[1]These operators are the Java *and*, *or* logical operators without shortcut evaluation. For the sake of clarity we omit here the definition of the branch distance for other operators.

In order to completely specify the branch distance we need to define its value in the base case of the recursion, that is, for atomic conditions. The particular expression used for the branch distance in this case depends on the operator of the atomic condition. The operands of the condition appear in the expression. A lot of research has been devoted in the past to the study of appropriate branch distances in software testing. An accurate branch distance taking into account the value of each atomic condition and the value of its operands can better guide the search. In procedural software testing these accurate functions are well-known and popular in the literature. They are based on distance measures defined for relational operators like $<$, $>$, and so on [10]. One of our contributions in this work is the definition of a distance measure for the `instanceof` operator. We defer the definition of this distance to Section 3.

When a test case does not reach the branching condition of target branch we cannot use the branch distance as objective function. In this case, we identify the branching condition c whose value must first change in order to cover the target branch (critical branching condition) and we define the objective function as the branch distance of this branching condition plus the *approximation level*. The approximation level, denoted here with $ap(c, b)$, is defined as the number of branching nodes lying between the critical one (c) and the target branch (b) [16].

In this paper we also add a real valued penalty in the objective function to those test cases that do not reach the branching condition of the target branch. With this penalty, denoted by p, the objective value of any test case that does not reach the target branching condition is higher than any test case that reaches the target branching condition. The exact value of the penalty depends on the target branching condition and it is always an upper bound of the target branch distance. Finally, the expression for the objective function is as follows:

$$f_b(x) = \begin{cases} bd_b(x) & \text{if } b \text{ is reached by } x \\ bd_c(x) + ap(c, b) + p & \text{otherwise} \end{cases} \quad (3)$$

where c is the critical branching condition, and bd_b, bd_c are the branch distances of branching conditions b and c.

Nested branches pose a great challenge for the search. For example, if the condition associated to a branch is nested within three conditional statements, all the conditions of these statements must be true in order for the program flow to proceed onto the next one. Therefore, for the purposes of computing the objective function, it is not possible to compute the branch distance for the second and third nested conditions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMinn calls the *nesting problem* [9]), which forces us to concentrate on satisfying each predicate sequentially. In order to alleviate the nesting problem, the test case generator selects as objective in each loop one branch whose associated condition has been previously reached by other test cases stored in the coverage table. Some of these test cases are inserted in the initial population of the EA used for solving the optimization problem. The percentage of individuals introduced in this way in the population is called the *replacement factor* and is denoted by Rf. At the beginning of the generation process some random test cases are generated in order to reach some branching conditions.

2.2 Class Instantiation

For the problem we are facing, we need to generate complex objects. For this reason, we have developed a generator of objects of a given class. The capacity of generating objects is essential in OO software testing. For creating an instance of a given class our object generator randomly selects a constructor of the class and generates random values for its parameters. If any of the parameters is an object, it is in turn generated by using again the object generator recursively. The base case of the recursion is reached when all the arguments of a constructor are primitive types.

In this work the generator returns constant values in the case where primitive types are required. The reason is that we focus here on the `instanceof` operator, and all the programs used in the experiments are composed of conjunctions of `instanceof` conditions. For this reason the state of the objects is not relevant, only its class is relevant.

3. DISTANCE FOR INSTANCEOF

In this section we present our proposal of distance measure for the `intanceof` operator. This distance will be used in the base case of the branch distance definition (see Section 2.1) when the `instanceof` operator is found. Since our distance is based on the class and interface hierarchy, first we need to present the notation for referring to the classes, interfaces, and their relationship. In Java there is no multiple inheritance among classes, that is, a class can only *extend* one class. However, this does not hold for interfaces: one interface can extend several interfaces and one class can *implement* several interfaces. The natural way of representing the class and interface hierarchy is by means of a graph. Following the terminology proposed in the Java language specification [6], we use the term *reference type* to refer to a class or an interface.

Let us denote with $G_R = (R, E_R)$ the graph representing a hierarchy of reference types, where R is the set of reference types considered and $E_R \subseteq R \times R$ is the set of arcs. We call this graph the *hierarchical graph* of R. The set R can be composed of all the classes and interfaces accessible from a virtual machine or a subset of them. We will use the notation C_R and I_R to refer to the set of classes and interfaces in R, respectively. The set of arcs E_R is determined by the relationship between classes and interfaces in R in the following way. Let $r1, r2 \in R$ be two reference types, then $(r1, r2) \in E_R$ if and only if:

- $r1, r2 \in C_R$ and class $r1$ is a *direct superclass* of $r2$
- $r1 \in I_R$ and interface $r1$ is a *direct superinterface* of reference type $r2$

We must recall here that there is only one class in Java with no superclass: `Object`. This fact, together with the lack of multiple inheritance for classes, implies that the subgraph of G_R composed only by the set of classes C_R is a directed tree with the `Object` class in the root. We denote this subgraph G_{C_R}.

At this point we can define the value returned by the `instanceof` operator based on our definition of hierarchical graph. Let "o `instanceof` r" be an `instanceof` expression where o is an object of class c and r a reference type. This expression evaluates to true if and only if a walk exists in G_R from r to c.

Once we have defined the hierarchical graph for a set of reference types R we present now the definition of the distance d between one class c and a reference type r. This distance is the one used for comparing the two arguments of an `instanceof` operator. In this operator, the first argument is an object from which only its class c is used for the comparison. The second argument can be any reference type r. We distinguish two cases: when r is a class and when r is an interface.

If r is a class then c and r belong to the directed tree G_{C_R}. Let us call c' the deepest class in the directed tree that is ascendant of both c and r at the same time. The class c' could also be c or r. Since G_{C_R} is a directed tree there exists a unique walk from c' to c, denoted by $w_{c'\to c}$, and a unique walk from c' to r , $w_{c'\to r}$. We call the first walk *hierarchical walk* and the second one *approximation walk*. The distance between c and r is defined as:

$$d(c,r) = h|w_{c'\to c}| + a|w_{c'\to r}| \ \text{, if } r \in C_R \quad , \tag{4}$$

where h and a are the *hierarchical* and *approximation* constants that weight the length of the hierarchical and approximation walks, respectively. In order to satisfy an `instanceof` expression the length of the approximation walk must be zero. However, in this last case, the length of the hierarchical walk is irrelevant to the satisfaction of the `instanceof` expression. For this reason, in order to reflect the real impact of each walk in the distance we should weight the approximation walk with a higher value than the hierarchical walk. We will experiment with the weights in the experimental section.

When r is an interface we consider the distance from c to the concrete classes of C_R that implement interface r. Let $S_r \subseteq C_R$ be the set of concrete (not abstract) classes implementing r (a walk exists in G_R from r to any class of S_r). Then the distance from c to r is defined as:

$$d(c,r) = \min_{t \in S_r} d(c,t) \ , \tag{5}$$

where $d(c,t)$ is the distance between two classes defined above in Eq. (4).

Fig. 3 illustrates the computation of the distance between class `ArrayList` and interface `Set`. There are two concrete classes that implement the `Set` interface in our example: `TreeSet` and `HashSet`. The distance between class `ArrayList` and them is in both cases the same: $2a+2h$. Thus, the distance between interface `Set` and class `ArrayList` is $d = 2a + 2h$. If we set $a = 50$ and $h = 10$, then $d = 120$.

The computation of the distance defined in this section has complexity $O(a + h)$ in the case of two classes and $O((a + h) * i)$ in the case of a class and an interface, where a and h are the approximation and hierarchical distances (maximum values in the case of the class and the interface) and i is the size of the subset S_r. However, in order to reduce the cost of computing these distances, these computations can be made prior to the test case generation and stored in a table.

4. EVOLUTIONARY SOLVER

EAs [2] 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.

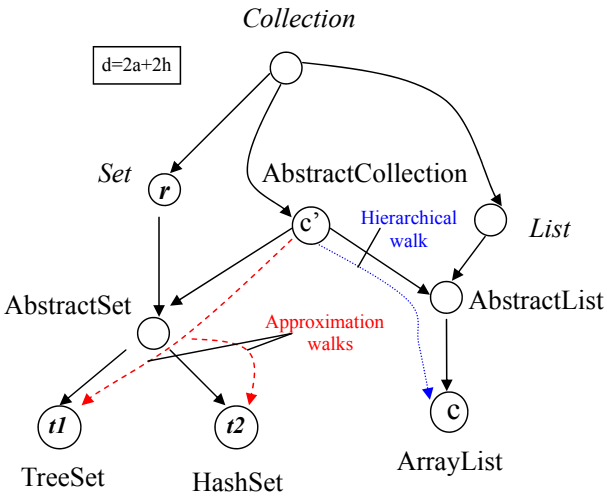

Figure 3: Example of distance between a class and an interface.

They are based on a set of tentative solutions (individuals) called *population*. The problem knowledge is usually enclosed in an objective function, the so-called *fitness function*, which assigns a quality value to the individuals.

Initially, the algorithm creates a population of μ individuals randomly or by using a seeding algorithm. At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation in order to compute a set of λ descendant individuals P'(t). The objective of the selection operator is to select some individuals from the population to which the other operators will be applied. The recombination operator generates a new individual from several ones by combining their solution components. This operator is able to put together good solution components that are scattered in the population. On the other hand, the mutation operator modifies one single individual and is the source of new different solution components in the population. The individuals created are evaluated according to the fitness function. The last step of the loop is a replacement operation in which the individuals for the new population $P(t+1)$ are selected from the offspring $P'(t)$ and the old one $P(t)$. This process is repeated until a stop criterion is fulfilled, such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target quality.

In the following we focus on the details of the specific EAs used in this work to perform the test case generation. Regarding the representation, one solution is a vector $\vec{o}$ of objects (see Fig. 4). These objects are used in the order determined by the vector as actual parameters of the method under test.

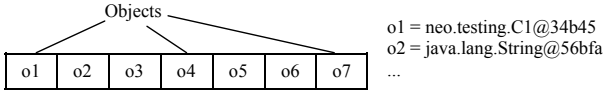

Figure 4: Representation of one solution vector $\vec{o}$.

For the selection operator we use q-tournament selection, which selects a single individual by choosing q individuals randomly from the population and selecting the best from

these q to survive. The recombination operator used is a single point crossover that works by cutting the chromosomes of the parents at some randomly chosen common point and then the resulting sub-chromosomes are swapped. We defer the description of the mutation operator to Section 4.1 because it is one of the contributions of this paper. The replacement operator is an elitist procedure, where any individual can potentially leave the pool and be replaced by a new one if the new one has a better fitness value. The fitness function used in this work is the one presented in Eq. (3).

4.1 Mutation Operator

The mutation operator decides whether or not to change a component of an individual according to the probability of mutation M. The mutation operator is usually designed to introduce small variations in the tentative solutions. In this work we propose the use of the distance defined in Section 3 for computing the probability to change one object of the solution to an instance of a different class. We call this mutation operator *distance-based mutation*, denoted by MDn, and it works as follows. For each component in the solution it decides whether to change it or not with probability M. If the object is changed, it selects the class of the new object from a universe U of concrete classes. If the old object had class c, the probability of changing to an object of class c' is given by the following expression:

$$p(c,c') = \begin{cases} \frac{\frac{1}{d(c,c')}}{\sum_{r \in U, r \neq c} \frac{1}{d(c,r)}} & \text{if } c \neq c' \\ 0 & \text{if } c = c' \end{cases} \quad (6)$$

where d is the distance between classes defined in Eq. (4). This way, it is more probable to mutate the object to a new one whose class is near the old one in the class hierarchy. This is how a "small change" is interpreted (and implemented) in the mutation operator. In Section 5.5, we will justify the use of an adaptive mutation MDα that evolves during the search. We defer its definition to the next section because its mathematical expression can be better understood after some results are shown.

5. EXPERIMENTAL SECTION

In this section, we apply the distance and mutation operator presented throughout the paper to a benchmark set of object-oriented test programs. Our main purpose is to study the measure of distance defined in Section 3 for the `instanceof` operator and the specific mutation operator designed. In the following section we first describe the test programs used for the experiments. We justify the parameterization of the algorithm in Section 5.2 and analyze the influence on the results of some parameters in Section 5.3. In Section 5.4 we compare the proposed mutation operator against a random mutation. Finally, in Section 5.5 we present the results of an adaptive mutation MDα, whose development has been motivated by the results of Section 5.4.

5.1 Test Programs

We use a benchmark set of nine test programs with different features available at `http://neo.lcc.uma.es`. Since we are interested in studying our proposals for dealing with the `instanceof` operator, all the atomic conditions in the test programs are `instanceof` expressions. This way we analyze the `instanceof` operator in isolation, avoiding any influence on the results of the distance expressions used for other relational operators. All the programs have the same number of branches and we use the name `obj`i_j to refer to the program with i atomic conditions per logical expression (varying from 2 to 4) and nesting degree j (varying from 1 to 3). Each program consists of one method with six conditions, varying the number of atomic conditions that appear in each one. In addition to this benchmark set, we use another program with only one condition composed of a conjunction of four `instanceof` expressions that will be used in an experiment discussed in Section 5.4.

5.2 Parameters of the algorithm

In the next section we show some preliminary results performed in order to set the best values for the parameters. However, in this section we want to specify and justify the parameters used in the experiments.

First, we start with the parameters for the proposed distance measure d. In this distance, a length for the approximation walk greater than zero implies that the `instanceof` expression is false. Thus, the approximation constant a must be large. The values used for a are 50, 100 and 200. The hierarchical constant h appears in the distance expression even when the `instanceof` expression is true. This distance helps the algorithm to search for test cases that are in the boundary of the satisfaction region of the logical expressions. It seems reasonable to think that this constant should be smaller than a and, for this reason, we use the values 1, 25, 50, and 100 for it. Regarding the parameters of the EA, we use the values 0.1, 0.2, and 0.3 for the probability of mutation M, and the values 0.25, 0.50, 0.75 for the replacement factor Rf presented in Section 2.1.

Since we are working with stochastic algorithms, we perform in all the cases a minimum of 30 independent runs of the algorithms (increased up to 200 in some cases) and Kruskal-Wallis's statistical tests for multiple comparison with a confidence of 95% (not shown for space reasons). In order to obtain well-founded conclusions we base all our claims on statistically significant differences.

5.3 Setting Parameters

In this section we analyze the influence on the average coverage of the replacement factor Rf, the mutation probability M, the approximation constant a, and the hierarchical constant h. The objective of this first study is to discover the best values for these parameters. The experiments performed have a factorial design. That is, for each of the nine test programs and each of the 108 combinations of the four above mentioned factors we have performed 30 independent runs. This means a total number of almost $30,000$ independent runs of the test case generator.

In this section we only show the average results of the most complex programs (`obj4_3`, `obj4_2`, and `obj3_3`) because the others achieve 100% coverage in most cases and could inflate coverage percentage. For each combination of Rf and M we have computed the average value of the coverage when the other parameters (a and h) change. In this way we have obtained Table 1. If we focus on the parameter M we can observe that the coverage is higher when M is small. The statistical tests confirm that the differences between the results obtained with $M = 0.1$ and the other two values are significant. Thus, we conclude that a small probability

Table 1: Average coverage percentage obtained changing M and Rf in the most complex programs

	$M = 0.1$	$M = 0.2$	$M = 0.3$
$Rf = 0.75$	83.08	76.10	67.29
$Rf = 0.50$	**83.43**	75.43	67.20
$Rf = 0.25$	82.10	73.81	66.74

Table 2: Average coverage obtained changing h and a in the most complex programs

	$h = 1$	$h = 25$	$h = 50$	$h = 100$
$a = 200$	75.45	75.33	74.93	75.68
$a = 100$	75.53	74.74	75.10	74.79
$a = 50$	74.86	75.81	74.44	**73.57**

of mutation must be used for programs with a high degree of nesting and a large number of atomic conditions in each logical expression. Regarding the replacement factor Rf, we can observe that the coverage increases with Rf when $M = 0.2$ and $M = 0.3$. However, the differences in the results are not statistically significant.

We have also studied the influence on the average coverage of the hierarchical and approximation constants h and a. As in the previous tables, for each program and each combination of a and h we have computed the average value of the coverage percentage when the other parameters (M and Rf) change. The results are shown in Table 2. On the basis of our previous intuition on the behaviour of the distance proposed for the `instanceof` operator we expected no significant differences except in the case $a < h$. The results confirm our expectations; as we can observe in Table 2, when $a < h$ the average coverage is minimum (with statistically significant differences). However, we cannot find a configuration that is better than all the rest. We can only conclude that a should not be less than h.

According to the results shown in this section we fix the values of the four parameters studied in the following experiments. The values chosen are $M = 0.1, Rf = 50\%, a = 50$ and $h = 25$. In the rest of the experimental section we analyze the proposed mutation operators.

5.4 Uniform vs. Distance-based Mutation

In this section we compare the distance-based mutation MDn against a simpler mutation operator: one that selects the class using a uniform distribution of probabilities. We will call this operator *uniform mutation* (MU) in the following. The only difference between the distance-based and the uniform mutation is the probability distribution used for selecting the new class c'. In MDn is given by Eq. (6) and in MU is given by:

$$p(c, c') = \begin{cases} \frac{1}{|U|-1} & \text{if } c \neq c' \\ 0 & \text{if } c = c' \end{cases} \quad (7)$$

where U is the universe of classes. One of the first questions we want to answer in this experimental section is: which mutation operator is better? Before the empirical evaluation we present a theoretical analysis that allows us to make some speculations. These speculations are based on the probability of obtaining an objective vector $\vec{t}$ from a given vector $\vec{o}$ by one application of the mutation operator. In the case of the uniform mutation this probability is

$$p_u(\vec{o}, \vec{t}) = (1 - M)^l M^{n-l} \left(\frac{1}{|U| - 1} \right)^{n-l} \quad (8)$$

where M is the mutation probability, n is the length of the vector $\vec{o}$, and l is the number of objects of $\vec{o}$ that are instances of the correct class (the class required by the objective vector $\vec{t}$). In the case of the distance-based mutation the expression is more complex:

$$p_d(\vec{o}, \vec{t}) = (1 - M)^l M^{n-l} \prod_{\{i | o_i \neq t_i\}} p(o_i, t_i) \quad (9)$$

where p is the probability distribution defined in Eq. (6) and we have used o_i and t_i to denote the classes of the corresponding objects.

At this point we must notice the following fact. If the classes of two objects o_i and t_i are near in the hierarchy of classes we have $p(o_i, t_i) > 1/(|U| - 1)$. Thus, $p_d(\vec{o}, \vec{t}) > p_u(\vec{o}, \vec{t})$ if solution $\vec{o}$ is near the objective $\vec{t}$. On the other hand, $p_d(\vec{o}, \vec{t}) < p_u(\vec{o}, \vec{t})$ if solution $\vec{o}$ is far from $\vec{t}$.

In order to check this speculation we have performed an experiment in which we used one program with one branch. The EA is used to search for a solution satisfying the condition. We have performed 200 independent runs of the EA and we have registered the best fitness of the population at each step of the EA in order to analyze the evolution of the search. In Fig. 5 we show the average of the 200 independent runs at each step of the execution for MDn and MU.

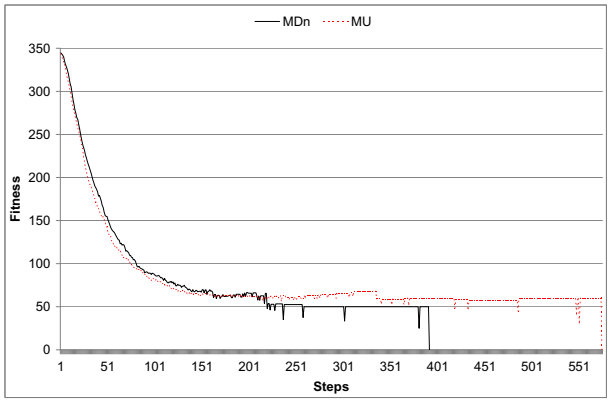

Figure 5: Fitness evolution with a uniformly initialized population.

We can observe in Fig. 5 that, although the behaviour is quite similar using the two mutation operators, the MU curve is lower than the other one at the beginning. The advance produced by MU to the objective solution is faster, since the initial population is far from this objective solution, so using MDn provides no advantage. However, as the search progresses we can observe that MDn is able to reach the objective solution before MU. When any individual of the population is near the objective solution, MDn guides the individual to the objective solution better than MU. This is the result expected from the speculations we made at the beginning of this section.

In order to confirm this behaviour we perform a new experiment in which the initial population is randomly generated using individuals that are near the objective solution.

We want to check if MDn is really faster than MU in this situation. In Fig. 6 we show the average evolution over 200 independent runs and we can observe that MDn has a clear advantage over MU when the population is near the optimum. Thus, our main conclusion in this section is that, in general, MDn is better than MU. However, MU can advance faster to the objective than MDn at the beginning of the search. Ideally, we would like to obtain a combination of the variability provided by the behaviour of MU at the beginning and the behaviour of MDn at the middle stage of the search. One way to achieve this is by means of an adaptive mutation operator. This is what we analyze in the following section.

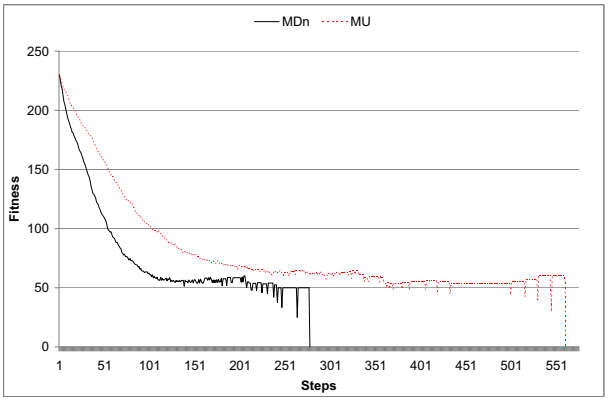

Figure 6: Fitness evolution with a population near the objective solution.

5.5 Adaptive Mutation

Motivated by the results of the previous section we present in this section a new mutation operator that changes its behaviour throughout the search. The difference between this adaptive operator, denoted by MDα, and the ones studied in the previous section is the probability distribution used for selecting a class. In MDα the probability distribution is:

$$p(c,c') = \begin{cases} \frac{\left(\frac{1}{d(c,c')}\right)^{\alpha}}{\sum_{r \in U, r \neq c} \left(\frac{1}{d(c,r)}\right)^{\alpha}} & \text{if } c \neq c' \\ 0 & \text{if } c = c' \end{cases} \quad (10)$$

In this expression, if $\alpha = 0$, we have the uniform mutation MU and if $\alpha = 1$, we have the distance-based mutation MDn. We can use values higher than 1 for α. If the value of α is high, then the mutation only selects classes that are close to the ones in the individual. We can see α as an exploitation-exploration parameter. A low value for α leads to an explorative search. A high value leads to an exploitative search. In order to make it adaptive we must change the value of α throughout the search. We use a linear increase for α, that is:

$$\alpha = \lambda \cdot step \quad (11)$$

where λ is a parameter called *adaptive speed*. With this expression for α, the behaviour of the adaptive mutation is the same as the behaviour of MU at the beginning and it switches to the behaviour of MDn as the search progresses. The higher the value of λ, the higher the speed of this change in the behaviour. If $\lambda = 0$ we have the uniform mutation, MU. On the other hand, if $\lambda = 1/T$, then MDα behaves like MDn in T steps.

The adaptive speed λ is a new parameter and we must analyze the behaviour of the algorithm for different values of λ in order to give some guidelines for selecting its value. A low value for λ means a very explorative search. A high value for λ makes the algorithm change very fast from the explorative phase to a very exploitative one. It is well-known in the metaheuristic field that one of the key points in the design of an algorithm is to select the exact balance between exploration and exploitation. Thus, we expect the best value for λ to be not too high and not too low: it should be something in between. In order to support these hypothesis we have applied our test case generator using the adaptive mutation to the nine programs presented in Section 5.1. We used nine different values for λ and performed 100 independent runs for each program and configuration. In all the cases the generator was executed until 100% branch coverage was obtained and we use the number of evaluations for comparison purposes. In Fig. 7 we show the average number of evaluations for all the programs and the nine values of λ. We have also included the results of MU ($\lambda = 0$).

As expected, when extreme values for λ are used the effort required to reach the total coverage is higher. In particular, when random mutation is used ($\lambda = 0$) the effort is higher than for intermediate values of λ (there are statistically significant differences that confirm this observation). On the other hand, when $\lambda = 1/60$, the higher value of λ, the effort required is again increased. The reason is that the search reaches a very exploitative stage in a few steps, in which newly generated solutions are similar to the parent solutions. In this situation it is difficult for the algorithm to reach the objective. The best values for λ are between 1/100 and 1/200.

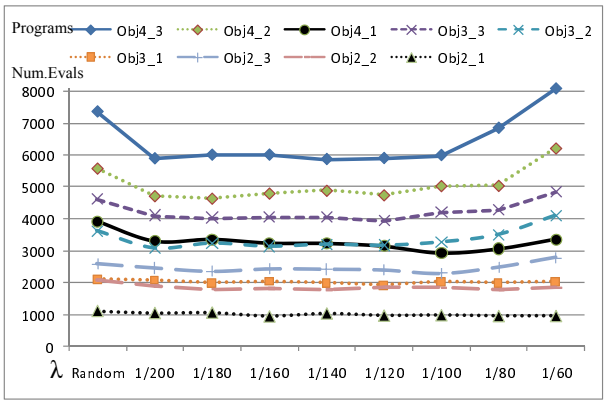

Figure 7: Average number of evaluations required for 100% branch coverage in all the test programs for different values of λ.

We have also compared our proposals against a random search. The random search proposes random classes for the vector of objects that is used as test case. The random search is able to reach 100% branch coverage only in `obj2_1` with an average of 1302 evaluations. For the other (more complex) programs we stopped the random search after 50,000 evaluations and the average coverage obtained

varies from 21% in `obj4_3` to 99% in `obj3_1`. In the results shown in Fig. 7 the maximum number of average evaluations for a 100% branch coverage is around 8,000. Thus, we conclude that our proposals are much better than a simple random search.

The results of Fig. 7 also show the "difficulty" of the programs for the test case generator. From the results we can sort the programs according to the effort required to reach 100% branch coverage. We can observe that, except in a few cases, this ranking is independent of the value of λ and is correlated with the value $i + j$ where i is the number of atomic conditions per logical expression, and j is the nesting degree. Furthermore, we can observe that the influence of λ on the results is higher in the "most difficult" programs, as we could expect.

6. CONCLUSIONS

In this paper we have focus on one aspect of OO Software, inheritance, to propose some approaches that can help to better guide the search of test cases in the context of OO evolutionary testing. In particular, we have proposed a distance measure to compute the branch distance in the presence of the `instanceof` operator in Java programs. We have also proposed two mutation operators that change the solutions based on the distance measure defined. In addition to the proposals we have performed a set of experiments to test our hypothesis. First, we have analyzed the most important parameters of the algorithm in order to select the best configuration. After that, we have analyzed and compared one of the proposed mutation operators against a uniform mutation. Finally, we have proposed an adaptive mutation operator that is able to make a better exploration and we have studied its main parameter.

As future work we plan to advance in the analysis of our proposals by performing theoretical studies on their behaviour. These analyses can help in selecting the values for the parameters of the mutation operators and the distance measure. In this work we have studied the `instanceof` expressions in an isolated way. In the future we plan to analyze how the distance measure defined and the mutation operators proposed can be combined with all the other approaches related to OO software testing. In addition, an analysis of the impact of the proposal in real-world software is a priority in our research plans.

7. ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). It has also been partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

8. REFERENCES

[1] Andrea Arcuri and Xin Yao. Search based software testing of object-oriented containers. *Inf. Sci.*, 178(15):3075–3095, 2008.

[2] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.

[3] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.

[4] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.

[5] Eugenia Díaz, Raquel Blanco, and Javier Tuya. Tabu search for automated loop coverage in software testing. In *Proceedings of the International Conference on Knowledge Engineering and Decision Support (ICKEDS)*, pages 229–234, Porto, 2006.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison Wesley, third edition, 2005.

[7] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of ob ject oriented programs. In *Proceedings of the Sixth Metaheuristics International Conference*, Vienna, August 2005.

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

[9] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *Proceedings of the Third UK Software Testing Workshop (UKTest 2005)*, pages 165–182, 2005.

[10] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[11] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[12] Paolo Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, 2004.

[13] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.

[14] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In *IEEE Congress on Evolutionary Computation (CEC)*, page 851–858, 2006.

[15] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*, page 1925–1932, 2006.

[16] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.

[17] Yuan Zhan and John A. Clark. The state problem for test generation in simulink. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1941–1948. ACM Press, 2006.