# Object distance and its application to adaptive random testing of object-oriented programs

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, Bertrand Meyer
Chair of Software Engineering
Department of Computer Science
ETH Zurich
CH-8092 Zürich
{Ilinca.Ciupa, Andreas.Leitner, Manuel.Oriol, Bertrand.Meyer}@inf.ethz.ch

## ABSTRACT

Testing with random inputs can give surprisingly good results if the distribution of inputs is spread out evenly over the input domain; this is the intuition behind Adaptive Random Testing, which relies on a notion of "distance" between test values. Such distances have so far been defined for integers and other elementary inputs; extending the idea to the testing of today's object-oriented programs requires a more general notion of distance, applicable to composite programmer-defined types.

We define a notion of *object distance*, with associated algorithms to compute distances between arbitrary objects, and use it to generalize Adaptive Random Testing to such inputs. The resulting testing strategies open the way for effective automated testing of large, realistic object-oriented programs.

## 1. OVERVIEW

One of the central issues of software testing is test cases selection. Effective testing increasingly requires strategies for *automatic* test case selection, since manual selection (for which a need will always remain) can only produce a subset of the large test suites that modern computing technology allows us to run. Counter to what intuition suggests, *random* strategies for selecting test inputs have proved remarkably effective when they can use a distribution of inputs that is spaced evenly over the range of possible values. The family of testing strategies called *Adaptive Random Testing* [4] is based on this idea. They are one of the most promising directions of automatic test case generation. Section 2 summarizes the contributions of this approach.

Work on Adaptive Random Testing has so far only considered inputs of primitive types such as integers, for which the notion of "evenly spaced" immediately makes sense: any such input belongs to a known interval on which there exists a total order relation. To test today's object-oriented programs, we also need to consider inputs that are composite objects with many fields, which exist during execution as a result of instantiating classes. We feel that it is desirable to extend the attractive concept of Adaptive Random Testing to such inputs; this paper describes an adaptive random strategy for selecting objects used in test cases for object-oriented programs.

The basic problem is, given a set of instances of various types, to select a subset for inclusion in a test suite so as to maximize the chances of finding a bug. Because multiple-field objects are not members of a totally ordered set, there's no notion of "equally spaced" inputs in a range. To address this issue, we rely on a notion of *object distance* to determine how far an object is "spaced" from another, and use it as a basis for object selection strategies.

The main contributions of this work include:

- A comprehensive notion of object distance (Section 3), which takes into account the various properties of an object: its dynamic type, which in O-O programming need not be identical to the type of another object to which we need to compute the distance (as both objects might be contenders for a call to the same routine[1] in the context of dynamic binding, both of their types inheriting from a common ancestor); the values of its primitive fields (integers, reals etc.); but also the *reference* fields that it may contain, leading to other objects and enabling the distance to take account of the structure of the run-time object graph.
- Strategies for Adaptive Random Testing (Section 4) based on the object distance and making it possible to select objects from a given set so as to maximize the diversity of objects in a test suite.
- A case study (Section 5), suggesting that the strategy does enhance that diversity and leads to effective testing.

[1]Another name for routines is "methods". Throughout the paper we use Eiffel-like terminology and notations.

Section 6 discusses various issues and Section 7 draws conclusions and outline directions for future work.

This work is part of a general project to produce test cases entirely automatically on the basis of contracts equipping object-oriented code, in particular reusable components. The testing IDE resulting from this project, AutoTest [6], is already in wide use and has uncovered bugs in a number of existing production systems. The testing strategies based on the notion of object distance developed in this paper are in the process of being integrated into AutoTest.

## 2. ADAPTIVE RANDOM TESTING

Random testing presents several benefits in automated testing processes. Some of its main benefits are ease of implementation, efficiency of test case generation, and the existence of ways to estimate its reliability [9]. Many reference texts are, however, critical towards it as the position of Glenford J. Myers [14] deems it the poorest testing methodology and "at best, an inefficient and ad hoc approach to testing".

Several studies [7, 8] disproved this assessment by showing that random testing can be more cost-effective than partition testing. Menzies and Cukic [13] conducted a simulation of random testing of programs whose structures and sizes varied widely, and came to the conclusion that random testing, even with only a few tests, is very useful. Andrews *et al.* [1] show that, when specific recommended practices are followed, a testing strategy based on random input generation finds bugs even in mature software, and does so efficiently. They also state that, in addition to lack of proper tool support, the main reason for the rejection of random testing is lack of information about best practices.

Several algorithms have therefore been developed which attempt to maintain the benefits of random testing while increasing its efficiency. They generally provide ways to guide testing, so that it is no longer purely random. Particularly promising is the family of algorithms developed around the seminal work of Chen *et al.* on Adaptive Random Testing (ART) [4]. ART is based on the intuition that an even distribution of test cases in the input space allows finding bugs through fewer test cases than with purely random testing. The implementation of ART requires keeping two disjoint sets of test cases: a *candidate* set and an *executed* set. The test cases in the candidate set are generated randomly. The executed set is initially empty; then, as testing progresses, test cases that are executed are added to it and removed from the candidate set. The first test case that gets executed is selected at random from the candidate set and is added to the executed set; in the subsequent steps, the test case from the candidate set that is furthest away from the executed ones is selected from the candidate set. The distance between two test cases is computed using the Euclidean measure: thus, for an n-dimensional input domain, the distance between two test cases $a$ and $b$ whose inputs are $a_i$ and $b_i$ respectively, for $i \in \{1, ..., n\}$, is $dist(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$. To evaluate the efficiency of ART, the authors use the "F-measure": the expected number of test cases required to reveal the first bug. Their experimental results show that ART can be more efficient than random testing by more than 50%.

Based on the ART intuition, a series of related algorithms have been proposed. Mirror ART [3] (MART) and ART through dynamic partitioning [5] reduce the overhead of ART. In MART, the input space is partitioned into disjoints subdomains. Test cases are generated in only one of these subdomains, using the ART algorithm, and then these generated test cases are mirrored into the other subdomains. This reduces the number of distance calculations that ART must perform. Restricted Random Testing [2] (RRT) is also closely related to ART and is based on restricting the regions of the input space where test cases can be generated. As opposed to ART, where the elements of the candidate set are generated randomly, in RRT test cases are always generated so that they are outside of the exclusion zones (a candidate is randomly generated, and, if it is inside an exclusion zone, it is disregarded and a new random generation is attempted). Further improvements to ART are provided by Mayer's work on lattice-based ART [12] and ART by bisection with restriction [11].

All the above algorithms need a measure of the distance between two test cases, which is calculated based on the distances between their integer (or real) inputs. If these inputs are not numeric, the algorithms cannot be applied. To increase the applicability of ART to object-oriented systems, we need a method for computing a measure of the degree of dissimilarity between two objects: which we refer to as the distance between them. Having such an object distance, as developed below, enables random testing of software written in an OO language to benefit from all the advantages provided by ART and its derivatives. Furthermore, the way this distance is calculated (the use of both a distance between types and a distance between values, and of different weightings for them) allows for high flexibility in implementing various testing strategies.

## 3. OBJECT DISTANCE

There are many ways to define a notion of distance between two objects. It is important to specify a framework for acceptable definitions, then make explicit any choices behind a specific proposal within that framework, and justify them. This discussion starts with a very general framework and makes a number of such choices until it arrives at a directly implementable notion, with an associated algorithm.

**Distance principle**: Inter-object distance should be a distance.

This refers to the mathematical notion of distance, i.e. a function $\leftrightarrow$ returning a real value between two objects $p$ and $q$ such that:

- $p \leftrightarrow q \geq 0$
- $p \leftrightarrow q = 0 \Leftrightarrow p = q$
- $p \leftrightarrow q = q \leftrightarrow p$
- $p \leftrightarrow q \leq p \leftrightarrow r + r \leftrightarrow q$ (the triangle inequality)

One of the consequences of the distance principle is that the rest of this discussion need only concern itself with defining

the distance between two distinct objects; the distance from an object to itself will be zero.

The basic issue is to define the distance between two composite objects $p$ and $q$. As illustrated in figure 2, each object is characterized by a number of fields, where each field is either:

- A directly usable **expanded** value: integer, boolean etc.
- A **reference** to another composite object.

We will include strings in the first case; although in many object-oriented frameworks a string value is denoted by a reference to an object containing the string's representation, it is more appropriate, when defining the distance, to treat the string as a directly usable value. Specifically, we will use as distance between two strings their *Levenshtein distance*, also known as edit distance: the minimum number of operations yielding one from the other where each operation is one of: substitution, insertion and deletion.

In the second case, the reference can be `void` (or `null`). To avoid special cases we will treat a void reference as a reference to a special object called *Void*.

## 3.1 Elementary distance

A basic measure is needed to compare elementary values as appears in fields of objects.

**Definition: elementary distance.** The distance between two elementary values $p$ and $q$ is:

- For numbers: $\mathbf{C}(|p - q|)$ (where $|p - q|$ is the absolute value of their difference and $\mathbf{C}$ is a monotonically non-decreasing function, with $\mathbf{C}(0) = 0$).
- For booleans: 0 if identical, $\mathbf{B}$ otherwise.
- For strings: the Levenshtein distance.
- For references: 0 if identical, $\mathbf{R}$ if different but none is void, $\mathbf{V}$ if only one of them is void.

In this definition, $\mathbf{B}$, $\mathbf{R}$ and $\mathbf{V}$ are positive values chosen conventionally; fully defining the model will also require defining an appropriate function $\mathbf{C}$. In Section 5 we will use $\mathbf{B} = \mathbf{R} = \mathbf{V} = 10$, and the identity function for $\mathbf{C}$ .

In the reference cases, it does not seem appropriate to compare the values of the references understood as addresses, since distance in memory usually carries no semantic relevance.

In the integer and string cases $\mathbf{C}$ is a compression function which can be used, if desired, to avoid giving too much weight to very large distances between integers or strings. Although integers and strings could use separate compression functions we use just one for simplicity. A possible choice for $\mathbf{C}$, other than the identity function, would be, for some constant $\mathbf{M}$, $\mathbf{C}(x) = x$ for $x \leq \mathbf{M}$ , and $\mathbf{C}(x) = \mathbf{M} + log_{10}(x - \mathbf{M} + 1)$ for $x > \mathbf{M}$.

## 3.2 Composite object distance

We may now turn to the issue of obtaining a proper definition of the distance between two composite objects $p$ and $q$. We should take into account the following three properties:

- A measure of the difference between their *types*. In object-oriented programming, any object is an instance of some class; that is one of its relevant properties, especially since a given (polymorphic) variable may at run time become attached to objects of different types, which in this case must have a common ancestor in the inheritance hierarchy. In comparing two objects we should estimate how far apart their types are in that hierarchy.
- A measure of the difference between the objects' individual *fields*, derived from their pair-wise elementary distances as defined above. We should compare these fields one by one, considering only "matching" fields corresponding to the same attributes in both objects; non-matching fields cause a difference but are captured by the type distance.
- For unequal references, a measure of the difference between the corresponding *objects*. This will be the same notion of object distance, applied recursively.

Any non-zero value for each of these three measures should increase the distance between $p$ and $q$; in other words the distance should be a monotonically non-decreasing function of each of its three components.

These requirements seem appropriate for any meaningful definition of distance between composite objects, leading to the following principle:

**Composite Object principle.** The distance between two distinct composite objects $p$ and $q$ is entirely determined by the following three values, and is monotonically increasing on each of them taken separately:

**Type distance:** A measure of the difference between the types of $p$ and $q$, independent of the values of the objects themselves.

**Field distance:** A measure entirely determined by the difference between the matching fields of $p$ and $q$.

**Recursive distance:** A measure entirely determined by the values of the distances between (recursively) objects $p.r$ and $q.r$, for all matching reference attributes $r$ such that $p.r \neq q.r$ (both non-`Void`).

We may express the Composite Object principle as a formula for the distance $p \leftrightarrow q$:

$$
\begin{aligned}
p \leftrightarrow q = \mathit{combination}(&\\
&\mathit{type\_distance}(p.\mathit{type}, q.\mathit{type}),\\
&\mathit{field\_distance}(p, q),\\
&\mathit{recursive\_distance}(\{[p.r \leftrightarrow q.r]\\
&\qquad | r \in \mathit{Reference\_attributes}(p.\mathit{type}, q.\mathit{type})\}))
\end{aligned}
\tag{1}
$$

where $Reference\_attributes(t1, t2)$ is the set of attributes of reference types applicable to both objects of type $t1$ and objects of type $t2$. We will look below at possible choices for the functions *combination*, *type_distance*, $field\_distance$ and $recursive\_distance$ will be specified next.

The last part of formula 1 is a recursive use of the distance function; for that reason we must treat formula (1) as a fixpoint equation, and ensure not only that the function combination is monotonic on each argument but also that a sequence $x_n = combination(a, b, x_{n-1})$ converges. A possible choice for $combination(a, b, x)$ is $a+b+\frac{1}{2}x$, but many others are available.

In the third clause of the Composite Object principle, we apply the distance recursively to objects obtained by following references. We need only consider reference pairs such that $p.r \neq q.r$ , since for equal references the object distance would be zero. In summing individual distances over a set of attributes $A$, we will always take the arithmetic mean (the sum divided by the number of its elements) to avoid penalizing objects that have large numbers of fields. The notation

$$\overline{\sum} a_i$$

will represent the arithmetic mean of the set of $a_i$.

## 3.3 Pair-wise field comparisons

The second and third clauses of the Composite Object principle indicate that we must only consider pair-wise differences between matching fields, corresponding to the same attribute, according to the following definition:

**Definition: matching fields.** Fields taken from objects $p$ and $q$ are matching if they are of the form $p.a$ and $q.a$ for an attribute $a$ common to their types, either because the types are the same or because they inherit a from a common ancestor.

Without this rule the comparison would make no sense. In particular, the name of the attributes is irrelevant: a class `PERSON` and a class `BOOK` may both have a field called *title*, but this does not indicate they are comparable.

## 3.4 Consistency requirements

The two distances depending on fields of the object, $field\_distance$ and $recursive\_distance$, should obey a consistency requirement:

**Field principle.** The field and recursive distances are entirely determined by pair-wise differences between the values of matching fields and objects (respectively), and are monotonically non-decreasing functions of each of these differences taken separately.

How these components of the distance depend on the corresponding fields will be expressed by the functions $field\_distance$ and $recursive\_distance$.

We may now look at possible choices for the functions that remain to be specified: *combination*, *type_distance*, $field\_distance$ and $recursive\_distance$.

## 3.5 Attaching weights to attributes

The last three functions cited serve to compare two objects or their types. Any composite object is made of a number of fields, each associated with an attribute (also called "data member" in C++) of the corresponding class. To define *type_distance* we must look at the attributes of the classes involved; for $field\_distance$ and $recursive\_distance$ we look at the fields of the objects.

In all three cases we shouldnt have to treat all attributes as contributing equally to the distance. Introducing a notion of weight gives us the necessary flexibility:

**Weight principle.** For every attribute of a class, it is possible to define an associated non-negative weight $w$, such that the type, field and recursive distances are monotonically increasing functions of $w * d$ for each applicable pair-wise distance $d$, and do not depend on $d$ if $w$ is 0.

This convention of writing the distance functions as functions of $w * d$ for each applicable elementary distance $d$ makes it possible to define a degree of relevance for various attributes of a given type and the corresponding fields, and to ignore certain fields altogether by giving them a zero weight.

It would be possible to define different weights for each of the type, field and recursive distances, but such extra flexibility does not seem warranted. Instead we just define, for each attribute of a class, how important the attribute is in comparing the corresponding objects.

The weight of an attribute $a$ will be written $weight_a$.

## 3.6 Type distance

The following rule guides the definition of the *type_distance*:

**Type distance principle.** The distance between two types is a monotonically increasing function of their path lengths to any closest common ancestor, and of the number of their non-shared features.

In this principle:

- A closest common ancestor two classes B and C is a class A that is an ancestor of B and C, and such that no proper descendant of A has this property (in other words in the figure we count A but not A).
- The path length from a class to an ancestor is the minimum number of nodes on a path to that ancestor, plus 1; in the figure the path length from C to A is 3.
- In languages where all types are based on a class and all classes have a common ancestor (`ANY`, `Object`), any two classes have a closest common ancestor. If this is not the case we will take the type distance to be infinite in the absence of a common ancestor.
- With multiple inheritance two types can have more than one closest common ancestor. In this case, the type distance must take into account distances to all of them.

- Non-shared features are features not inherited from a common ancestor. In the figure 2 *is_sick* is a non-shared feature; so is *name* in *PERSON* since the presence of a *name* feature in *PET* is just accidental homonymy.

The second part of the principle corresponds to the intuition that the more individual features differ, the more the corresponding objects should differ.

We will use the following as type distance for two types $t$ and $u$:

$$type\_distance(t,u) = \lambda * path\_length(t,u) + \nu * \overline{\sum}_{a \in non\_shared(t,u)} weight_a \quad (2)$$

here *path_length* denotes the minimum path length to a closest common ancestor, and *non_shared* the set of non-shared features. $\lambda$ and $\nu$ are two non-negative constants; in Section 5 we will choose 1 for both.

## 3.7 Combining fields

There remains to define the field and recursive distances in accordance with the above requirements. A simple choice for the field distance is:

$$field\_distance(p,q) = \overline{\sum_a} weight_a * elementary\_distance(p.a, q.a) \quad (3)$$

This is a sum over matching expanded attributes $a$, with the convention noted above for void references.

We may use a similar formula for the recursive distance:

$$recursive\_distance(p,q) = \overline{\sum_r} weight_r * (p.r \leftrightarrow q.r) \quad (4)$$

This is a sum over matching reference attributes $r$; as noted we need only consider the fields for which $p.r$ and $q.r$ are not equal and neither of them is *Void*.

We also use a simple additive formula for the combination of the three component distances:

$$combination(fd, td, rd) = \tau * td + \phi * fd + \alpha * rd \quad (5)$$

where $\alpha$ is an attenuation factor, between 0 and 1 (excluded), introduced to ensure convergence as discussed above; in the example from Section 5 we will use $\alpha = \frac{1}{2}$. $\tau$ and $\phi$ are non-negative constants.

The following formula gives the full distance definition combining the previous definitions:

$$\begin{aligned} p \leftrightarrow q = \\ \tau * \lambda * path\_length(p.type, q.type) \\ + \tau * \nu * \overline{\sum}_{a \in non\_shared(p.type, q.type)} weight_a \\ + \phi * \overline{\sum_a} weight_a * elementary\_distance(p.a, q.a) \\ + \alpha * \overline{\sum_r} weight_r * (p.r \leftrightarrow q.r) \end{aligned} \quad (6)$$

where $a$ ranges over all matching fields and $r$ over all matching non-equal, non-*Void* reference fields.

## 3.8 Parameters to the model

The distance definition has introduced a number of constants and a function, for which any application must choose values. Here is a recapitulation of all these parameters, indicating for each, in parentheses, the value chosen for the example presentation below.

**B** (10) : distance between two unequal boolean values.

**V** (10) : distance between a non-void reference and Void.

**R** (10) : distance between two unequal, non-void references.

**C** (identity function): compression function for integer and string distances.

$\lambda$ (1): in the type distance, weight of the path length part.

$\nu$ (1): in the type distance, weight of the part involving non-shared attributes.

$\tau$ (1): global weight for the type distance.

$\phi$ (1): global weight for the field distance.

$\alpha$ ($\frac{1}{2}$): attenuation factor (global weight for the part of the distance that depends on referenced objects).

$weight_a$ for every attribute $a$ (1): weight of $a$ (used for the type distance, the field distance and the recursive distance).

In the absence of a clear rationale for more specific values, the choices so far, as reflected in the application below, use simple values: 1 to assign equal weight to the various components of a distance, $\frac{1}{2}$ for the attenuation factor to ensure reasonably fast convergence, and 10 as a conventional measure of distance between two clearly different booleans or references. We do not have enough experience with the model to know whether these simple values suffice or whether we need more sophisticated tuning. The presence of all the parameters listed will make such tuning possible if turns out to be necessary.

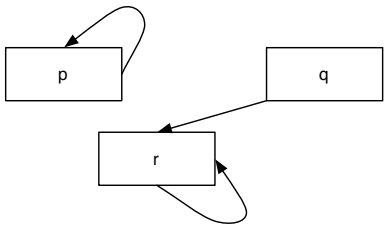


**Figure 1: Structure paradox**

## 3.9 The structure paradox

One effect of the general approach to distance measurement introduced here is that it pays more attention to actual values than to isomorphism of structures. With the example from Figure 1 and the attenuation factor $\alpha$ being set to $\frac{1}{2}$, we have $r \leftrightarrow p = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + ... = 1$

$r \leftrightarrow q = 0$

This reflects the property that the objects $q$ and $r$ are field-by-field identical, but $p$ and $q$ are not.

It is possible to argue instead that $p$ is more similar to $r$ than $q$ is since the corresponding deep object structures are isomorphic.

Both views are sustainable. The distance as defined here corresponds to notions of "shallow" equality, as implemented in Eiffel by `is_equal` and in Java by `equals`. A notion of distance corresponding to "deep equality", as in Eiffel's `is_deep_equal` function, seems also possible; it would need to estimate a distance between graph structures. This approach would have to use measures of structural distance between graphs, as developed for example in [15], and avoid the difficulties associated with the graph isomorphism problem. In the present discussion we stick to a more elementary notion of distance based on values of references rather than structural similarity.

## 3.10 A matrix form

Applying formula (6) to a set of mutually related objects gives a matrix fixpoint equation form, which facilitates both illustration and resolution.

Consider a set $P$ of objects $p_1$, $p_2$, , $p_n$. Let $D$ be the matrix of pair-wise distances between these objects:$D$ $[i, j]$ is the distance between $p_i$ and $p_j$. $D$ and all otehr matrices involved in this discussion are symmetric.

Let $T$ be the matrix of type distances between the elements of $P$, and $F$ the matrix of their field distances, as computed from their matching fields. Then we may rewrite formula 6 applied to all the points in $P$ as the equation:

$$D = \tau T + \phi F + \alpha * \overline{\sum}_{r \in References} D \circ r \qquad (7)$$

Where $D \circ r$ is the matrix $D'$ such that

$$\forall i, j \ D'[i, j] = weight_r * D[p_i.r, p_j.r]$$

with the convention that $p.r$ and the corresponding matrix entries are zero if the attribute $r$ is not applicable to $p$.

This last term of equation (7) simply results, for any particular reference attribute $r$, from applying the last term of equation (6).

$$weight_r * (p.r \leftrightarrow q.r)$$

to all objects: we replace every object $p$ by the object found by following the reference in the $r$ field of $p$. For the $[i, j]$ entry of the matrix $D$, which corresponds to the distance between objects $p_i$ and $p_j$, we apply this transformation to both $p_i$ and $p_j$.

The transformation induced by a reference attribute $r$ is generally not a substitution (one-to-one mapping) because of the possibility of dynamic aliasing (two references attached to the same object) and of void references (which we represent as references to a fixed object *Void*).

We may use equation (7) to compute the pair-wise distances between all the points in a set iteratively. With a small enough attenuation factor $\alpha$ it will quickly reach a satisfactory approximation.

For a precise computation of the distance, applying equation (7) assumes that the set $P$ of objects is *reference-complete*: any object reachable directly or indirectly from a member of $P$ by following references is also in $P$. The simplest reference-complete set is the set of all objects, but this will usually be too large to handle. Two reasonable policies are:

1. Start from a set $P$ that is not necessarily reference-complete, for example a set containing just two objects $p$ and $q$ of which we need to compute the distance and extend it repeatedly as application of equation (7) brings in the need to consider new objects.
2. Alternatively, work with a fixed set $P$, but whenever the application of equation 7 brings in an object not in $P$ take an arbitrarily chosen large value for its distance to any member of $P$.

# 4. DISTANCE-BASED TESTING STRATEGIES

Various testing strategies can be implemented using the computation of the distance between two objects presented in the previous section. For instance, ART is based on the intuition that spreading out values in the input domain will result in a decreased F-measure. One possible implementation of this concept would be to keep a set of candidate test inputs and a set of already used inputs, and always choose from the candidate set the value v that has the highest average distance to all values used before. Then v is removed from the candidate set and added to the used values set. The test case is executed and, if it passes (so no bug is found), the next best value is selected from the candidate set.

This algorithm is described by the following fragment of pseudo-code, where the `distance` function is implemented as described in the previous section:

```
used_objects: SET [ANY]
```

```
candidate_objects: SET [ANY]
current_best_value: INTEGER
v0, v1: ANY
current_accumulation: DOUBLE
...

current_best_value := candidate_objects.choose

foreach v0 in candidate_objects
do
   current_accumulation := 0.0;
   foreach v1 in used_objects
   do
      current_accumulation :=
         current_accumulation + distance (v0, v1);
   end
   if (current_accumulation > current_best_value)
   then
      current_best_value := v0;
   end
end
```

This strategy is similar to the one originally proposed for ART [4], the differences being the selection criterion (average distance rather than maximum minimum distance) and the computation of the distance measure. A number of variations are possible within that framework; in particular they can play with weights and the model's other parameters to emphasize or downplay specific properties of objects and types. Varying the number of fixpoint iterations is also a way to tune the model.

# 5. CASE STUDY

In this section we apply the testing algorithm introduced in the previous section on an example. We first introduce a class hierarchy and corresponding sample instances that will be used throughout this section. Then we pre-calculate all the necessary object distances using our proposed definition. The resulting object-distances will be used in the next step, which consists of stepping through the execution of the testing algorithm, where in each step the next object to be used for testing is decided.

We consider the classes depicted in Figure 2 and objects depicted in Figure 3. The example involves both reference and primitive types, and includes self-references, and polymorphism.

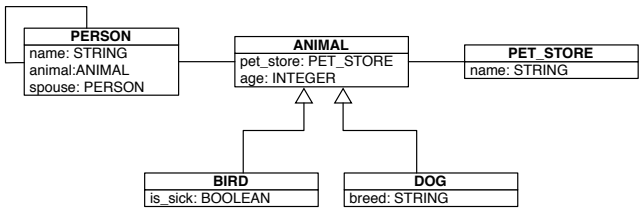


**Figure 2: Example class diagram**

We assume that the routine under test takes one argument which is of type $PERSON$. Our approach trivially extends to routines with multiple arguments; here we use one argument for the sake of brevity. In this section we will calculate the difference between all persons. A potential implementation will quite likely use a lazy strategy to only calculate those distances that the testing algorithm actually requires.

The following will show the necessary computations bottom-up starting with first type and field comparisons, then pet shop, animal and finally person comparisons.

The type distance between a type and itself is zero, hence the only interesting type comparisons in our example are between DOG and BIRD:

$$type\_distance(DOG, BIRD) = 2 + [1 + 1] = 4$$

The path_length between DOG and BIRD is 2 and there are two non-shared features (*is_sick* and *breed*, both weighted 1).

Next the differences between the strings occurring in the example are, according to the Levenstein distance:

$$\begin{array}{l} field\_distance(\text{"}Store1\text{"},\text{"}Store2\text{"}) = 1 \\ field\_distance(\text{"}Steve\text{"},\text{"}Mary\text{"}) = 5 \\ field\_distance(\text{"}Steve\text{"},\text{"}Jenna\text{"}) = 5 \\ field\_distance(\text{"}Steve\text{"},\text{"}Kelly\text{"}) = 5 \\ field\_distance(\text{"}Mary\text{"},\text{"}Jenna\text{"}) = 5 \\ field\_distance(\text{"}Mary\text{"},\text{"}Kelly\text{"}) = 4 \\ field\_distance(\text{"}Jenna\text{"},\text{"}Kelly\text{"}) = 4 \end{array}$$

The calculations for the two pet shops follow:

$$\begin{array}{rcl} ps1 \leftrightarrow ps2 & = & td(PET\_SHOP, PET\_SHOP) \\ & & +(fd(\text{"}Store1\text{"},\text{"}Store2\text{"})) \\ & = & 0 + 1 = 1 \end{array}$$

Since both *ps1* and *ps2* have the same type (PET_SHOP) their type distance is zero; the difference between their only field (name of type STRING) is 1 as already calculated.

The calculations for comparing the animals *bird1*, *bird2* and *dog1* to each other are shown below:

$$\begin{array}{rcl} bird1 \leftrightarrow bird2 & = & td(BIRD, BIRD) \\ & & +\frac{1}{2} * [|bird1.age - bird2.age| + 0] \\ & = & 0 + \frac{1}{2} * [2 + 0] = 1 \\ bird1 \leftrightarrow dog1 & = & td(BIRD, DOG) \\ & & +\frac{1}{2} * [|bird1.age - dog1.age| + \mathbf{R}] + \\ & & \frac{1}{2} * [ps1p \leftrightarrow s2] \\ & = & 4 + \frac{1}{2} * [7 + 10] + \frac{1}{2} * 1 = 13 \\ bird2 \leftrightarrow dog1 & = & td(BIRD, DOG) \\ & & +\frac{1}{2} * [|bird2.age - dog1.age| + \mathbf{R}] + \\ & & \frac{1}{2} * [ps1 \leftrightarrow ps2] \\ & = & 4 + \frac{1}{2} * [9 + 10] + \frac{1}{2} * 1 = 14 \end{array}$$

The type distance between *bird1* and *bird2* is zero since they have the same type. They share two fields, hence the sum of their field differences is divided by 2. Their first fields (*age*) are set to 3 and 1 respectively hence we get a difference of 2. Their second field (*pet_shop*) is for both set to *ps1* hence they do not differ. The type distance for *bird1* and *dog1* is 4 as their types are BIRD and DOG respectively. The sum of their field distances is again divided by 2, due to the 2 shared fields. The age difference is now $10 - 3 = 7$ and they come from different pet shops ($ps1 \leftrightarrow ps2 = 1$). The difference between *bird2* and *dog1* is calculated correspondingly.

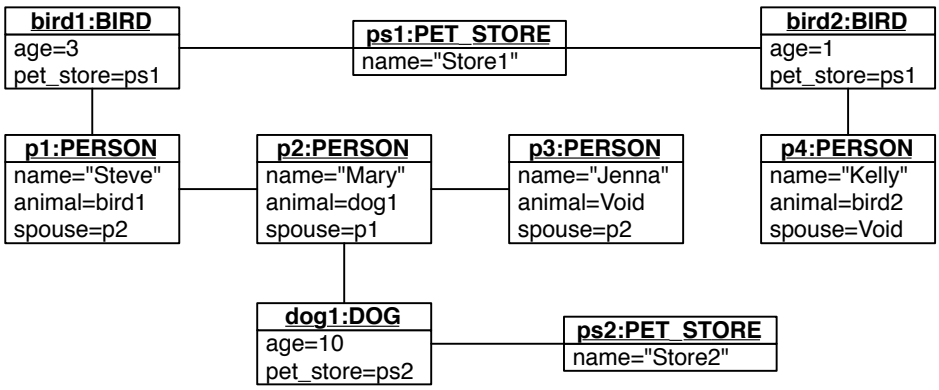


**Figure 3: Example object diagram**

The following symmetric matrix depicts the distances between the animals.

| | $bird1$ | $bird2$ | $dog1$ |
|---|---|---|---|
| $bird1$ | 0 | | |
| $bird2$ | 1 | 0 | |
| $dog1$ | 13 | 14 | 0 |

As a last step we compare the persons *p1*, *p2*, *p3* and *p4* to each other. Note that for brevity, we omit the type distance calculations for persons since the type distance between all persons is always zero (they are all of type PERSON).

$$\begin{aligned} p1 \leftrightarrow p2 &= \tfrac{1}{3} * [fd(\text{"}Steve\text{"},\text{"}Mary\text{"}) + \mathbf{R} + \mathbf{R}] \\ &\quad +\alpha * \tfrac{1}{2} * [bird1 \leftrightarrow dog1 + p1 \leftrightarrow p2] \\ &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + p1 \leftrightarrow p2] \end{aligned}$$

Objects *p1* and *p2* share three fields: *name*, *animal* and *spouse*. Hence the sum of their field differences is divided by 3. The difference between their names ("Steve" and "Mary") is 5 as calculated during the string comparisons. Both the attributes *animal* and *spouse* are distinct hence they weight $\mathbf{R} = 10$ each. The rest of the formula is due to the recursive distance. There is the attenuation factor $\alpha$ ($\frac{1}{2}$) and the normalization through the number of shared reference fields (*animal* and *spouse*). As calculated previously the difference between the animals $bird1 \leftrightarrow dog1 = 13$. The difference between the spouses is $p1 \leftrightarrow p2$. Note the introduction of recursion in this last step. The resulting fix-point equation is now solved iteratively (using an initial value of zero; we show 3 significant digits to illustrate speed of convergence):

$$p1 \leftrightarrow p2_{n+1} = \tfrac{25}{3} + \tfrac{1}{4} * [13 + p1 \leftrightarrow p2_n]$$

$$\begin{aligned} p1 \leftrightarrow p2_0 &= 0 \\ p1 \leftrightarrow p2_1 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 0] = 11.583 \\ p1 \leftrightarrow p2_2 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 11.583] = 14.479 \\ p1 \leftrightarrow p2_3 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 14.479] = 15.203 \\ p1 \leftrightarrow p2_4 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 15.203] = 15.384 \\ p1 \leftrightarrow p2_5 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 15.384] = 15.429 \\ p1 \leftrightarrow p2_6 &= \tfrac{25}{3} + \tfrac{1}{4} * [13 + 15.429] = 15.441 \end{aligned}$$

We abort the iteration after 6 steps and continue working with the approximation of 15.4 for $p1 \leftrightarrow p2$. Next the distance between *p1* and *p3*:

$$\begin{aligned} p1 \leftrightarrow p3 &= \tfrac{1}{3} * [fd(\text{"}Steve\text{"},\text{"}Jenna\text{"}) + \mathbf{V} + 0] \\ &\quad +\alpha * \tfrac{1}{1} * [p2 \leftrightarrow p2] \\ &= \tfrac{15}{3} + \tfrac{1}{2} * \tfrac{1}{1} * [0] \end{aligned}$$

Note that in the recursive step in the above formula there is no comparison between animals, because one of the animals is void. Since there is no recursion in the above equation we can evaluate the result immediately and get $p1 \leftrightarrow p3 = 5$

Next we compare objects *p1* and *p4*:

$$\begin{aligned} p1 \leftrightarrow p4 &= \tfrac{1}{3} * [fd(\text{"}Steve\text{"},\text{"}Kelly\text{"}) + \mathbf{R} + \mathbf{V}] \\ &\quad +\alpha * \tfrac{1}{1} * [bird1 \leftrightarrow bird2] \\ &= \tfrac{25}{3} + \tfrac{1}{2} * \tfrac{1}{1} * [1] \end{aligned}$$

This distance can again be evaluate directly: $p1 \leftrightarrow p4 = 8.8$ (rounded to one decimal).

Comparing objects *p2* and *p3* we start with:

$$\begin{aligned} p2 \leftrightarrow p3 &= \tfrac{1}{3} * [fd(\text{"}Mary\text{"},\text{"}Jenna\text{"}) + \mathbf{V} + \mathbf{R}] \\ &\quad +\alpha * \tfrac{1}{1} * [p1 \leftrightarrow p2] \\ &= \tfrac{25}{3} + \tfrac{1}{2} * [15.4] \end{aligned}$$

Since we have already calculated the distance between *p1* and *p2* we can evaluate the above equation directly and receive: $p2 \leftrightarrow p3 = 16.0$ (rounded to one decimal).

The second but last person comparison involves *p2* and *p4*:

$$\begin{aligned} p2 \leftrightarrow p4 &= \tfrac{1}{3} * [fd(\text{"}Mary\text{"},\text{"}Kelly\text{"}) + \mathbf{R} + \mathbf{V}] \\ &\quad +\alpha * \tfrac{1}{1} * [dog1 \leftrightarrow bird2] \\ &= \tfrac{24}{3} + \tfrac{1}{2} * \tfrac{1}{1} * [14] \end{aligned}$$

Hence we get $p2 \leftrightarrow p4 = 15$. And finally the last person comparison between *p3* and *p4*:

$$\begin{aligned} p3 \leftrightarrow p4 &= \tfrac{1}{3} * [fd(\text{"}Jenna\text{"},\text{"}Kelly\text{"}) + \mathbf{V} + \mathbf{V}] \\ &= \tfrac{24}{3} = 8 \end{aligned}$$

Combining all those calculation we receive the following matrix showing the difference between persons *p1*, *p2*, *p3*, and *p4*:

| | $p1$ | $p2$ | $p3$ | $p4$ |
|---|---|---|---|---|
| $p1$ | 0 | | | |
| $p2$ | 15.4 | 0 | | |
| $p3$ | 5.0 | 16.0 | 0 | |
| $p4$ | 8.8 | 15.0 | 8.0 | 0 |

Now that all distances between persons are known we can start testing. Initially no object has been used for testing yet. Set *used_objects* is empty and *candiate_objects* is $\{p1, p2, p3, p4\}$. In this state our algorithm will pick an arbitrary object from the *candidate_objects* set. Let us assume *p1* was picked.

After the first round of testing the set *used_objects* is $\{p1\}$ and the set *candidate_objects* is $\{p2, p3, p4\}$. According to the algorithm, the next object to pick for testing is the one from *candidate_objects* with the biggest distance to $p1$; in our case this is $p2$.

After moving $p2$ we now have $used\_objects = \{p1, p2\}$ and set $candidate\_objects = \{p3, p4\}$. The accumulated distance of $p3$ to $p1$ and $p2$ is $p3 \leftrightarrow p1 + p3 \leftrightarrow p2 = 21$ and the accumulated distance of $p4$ to $p1$ and $p2$ is $p4 \leftrightarrow p1 + p4 \leftrightarrow p2 = 23.8$. Hence, we next choose $p4$ for testing and move it from the *candidate_objects* set to the *used_objects* set.

After the third round of testing the set *used_objects* is $\{p1, p2, p4\}$ and the set *candidate_objects* is $\{p3\}$.

In the fourth round of testing only one candidate is left and will be chosen for testing. We have assumed throughout the case study that test cases using the selected values always pass, in order to demonstrate how the value selection proceeds. In a realistic setting, when a test case fails, testing will most likely stop and only re-start when the found bug is fixed.

The example we have shown seems to suggest that our algorithm picks the input objects in an order similar to that of an intuitive selection.

In this section we have first applied our measure to instances of some example classes involving primitive types, references, several levels of indirection, cycles and void references and polymorphism and then used the resulting distance values to drive the test data selection of a testing algorithm.

## 6. DISCUSSION

One of the issues we considered was whether we should use not only attributes in the calculation, but also argument-less functions. The results of such functions are obviously computed on the basis of attribute values; however, the existence of these functions indicates that they are representative for the semantics of the encompassing class, hence it may be reasonable to assume that a greater weight should implicitly be given to the attributes that they rely on. Although this argument speaks for including functions without arguments in the object distance calculation, there is one major problem: the execution of such a function might change the state of the object, and it is not acceptable that the distance calculation should trigger a state change – it must be side-effect free. Furthermore, what happens, in the case of languages which support executable specification such as Eiffel, if the precondition of such a function is not fulfilled?

Another alternative idea for our model is to use a developer-defined function that compares the content of two objects to decide if they are equal (an example of such a function in Java would be `equals`). If such a function returns `True` for two objects, it might be a reasonable assumption that we can set their value distance to 0. However, this would break the consistency of our distance calculation. Such a function can still be used in the implementation of the distance computation, when setting the value distance weights automatically (as opposed to using a user-provided weight vector).

The implementation of the object distance presented here is black-box; we assume no knowledge about the internal structure of the code under test or of its specification. A white-box approach where the testing goal is not sheer coverage of the input domain or of its partitions would bring entirely new aspects to our distance calculation. As a first example, if the testing goal is path coverage, we will assign the maximum value to the distance between any two inputs that cause different paths in the code to be executed. Such a strategy could be implemented (much in the spirit of the algorithms for using symbolic execution for testing developed by Khurshid et al. [10]) by first performing symbolic execution to gather the path conditions (conditions that the inputs must fulfill for a certain instruction in the code to be executed), and then using these conditions to calculate the object distance. Moreover, if the source code is available, we could monitor attribute accesses in the code under test, and, in the distance calculation, only use the attributes that are actually accessed.

Performance is a possible concern, including both the cost of the distance calculation itself and the number of such calculations needed by the testing strategy.

For the distance calculation, there is (as noted) no need to include all possible objects. We can use, as illustrated informally in the case study, a "lazy" approach: start with only the objects on which we need distance values in the end; then add referenced objects as the traversal of the object graph needs them, or ignore some altogether (simply giving them a high distance value) in a performance-precision tradeoff. Further analysis and experimentation of such strategies are in progress.

The problem of the number of distances to compute also exists in previous ART work, and the corresponding solutions should be applicable to the present framework.

## 7. CONCLUSIONS

We have described a model for representing the differences between two objects, and, based on this model, an algorithm for computing a value for the distance between them. We have also shown how this model can be used in various distance-based testing strategies.

The applications of the object distance potentially extend beyond the scope of this paper and the general field of testing. One example would be a data lookup system, where we search for objects that are similar enough (within a given distance range) to a prototype object. Another would be a data change tracking system, where we only want to record changes to the object state that exceed a given delta. Such applications must be explored further.

Further work on the notion of object distance can pursue

several directions. We first intend to run a wide-scale evaluation of its performance, by integrating it into the fully automated random testing framework AutoTest [6]. We will also investigate possible variants of the calculation algorithm described here. Furthermore, a white-box approach would allow for a very wide variety of implementations for the object distance, based on the testing target (code coverage, etc). Finally, various distance-based testing strategies extending the basic idea illustrated in Section 4 can be developed.

## Acknowledgements

We thank Stephanie Balzer, Lisa (Ling) Liu and Bernd Schoeller for many helpful discussions and very useful comments on the work presented here.

## 8. REFERENCES

[1] J. H. Andrews, S. Haldar, Y. Lei, and C. H. Li. Randomized unit testing: Tool support and best practices. Technical Report 663, Department of Computer Science, University of Western Ontario, January 2006.

[2] K. P. Chan, T. Y. Chen, and D. Towey. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, pages 321 – 330. Springer-Verlag, London, UK, 2002.

[3] T. Chen, F. Kuo, R. Merkel, and S. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, pages 4 – 11, 2003.

[4] T. Chen, H. Leung, and I. Mak. Adaptive random testing. In M. J. Maher, editor, *Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making. 9th Asian Computing Science Conference. Proceedings*. Springer-Verlag GmbH, 2004.

[5] T. Chen, R. Merkel, P. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Proceedings of the Fourth International Conference on Quality Software*, pages 79 – 86, 2004.

[6] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*, pages 545–557, September 19-22 2005.

[7] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10:438 – 444, July 1984.

[8] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16 (12):1402–1411, December 1990.

[9] R. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[10] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 553–568, 2003.

[11] J. Mayer. Adaptive random testing by bisection with restriction. In *Proceedings of the Seventh International Conference on Formal Engineering Methods (ICFEM 2005)*, LNCS 3785, pages 251–263. Springer-Verlag, Berlin, 2005.

[12] J. Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, ACM, pages 333–336. ACM Press, New York, NY, USA, 2005.

[13] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, September - October 2000.

[14] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[15] K. Zhang, J. T.-L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *CPM*, pages 395–407, 1995.