ARTOO: Adaptive Random Testing for Object-Oriented Software

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, Bertrand Meyer
Chair of Software Engineering
ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Intuition is often not a good guide to know which testing strategies will work best. There is no substitute for experimental analysis based on objective criteria: how many faults a strategy finds, and how fast. "Random" testing is an example of an idea that intuitively seems simplistic or even dumb, but when assessed through such criteria can yield better results than seemingly smarter strategies. The efficiency of random testing is improved if the generated inputs are evenly spread across the input domain. This is the idea of Adaptive Random Testing (ART).

ART was initially proposed for numerical inputs, on which a notion of distance is immediately available. To extend the ideas to the testing of object-oriented software, we have developed a notion of distance between objects and a new testing strategy called ARTOO, which selects as inputs objects that have the highest average distance to those already used as test inputs. ARTOO has been implemented as part of a tool for automated testing of object-oriented software.

We present the ARTOO concepts, their implementation, and a set of experimental results of its application. Analysis of the results shows in particular that, compared to a directed random strategy, ARTOO reduces the number of tests generated until the first fault is found, in some cases by as much as two orders of magnitude. ARTOO also uncovers faults that the random strategy does not find in the time allotted, and its performance is more predictable.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Verification

Keywords

software testing, adaptive random testing, object distance

ICSE'08, May 10–18, 2008, Leipzig, Germany.

1. OVERVIEW

Testing remains the most widely used method for assessing software quality. Recently, automated solutions have become increasingly practical. They do not, however, remove the basic limitation of testing: the impossibility of exhaustiveness for any non-trivial program, requiring testers to come up with strategies for selecting inputs to be tested in the time available.

One possible strategy is random testing. It has several advantages: wide practical applicability, ease of implementation in an automatic testing tool, no overhead for choosing inputs out of the set of all inputs, lack of bias. This absence of any system in choosing inputs is also what exposes random testing to the most criticism. Several other strategies for input generation have been proposed (symbolic execution combined with constraint solving [30], [18], direct setting of object fields [5], genetic programming [29], etc.), but none of these strategies reaches the level of applicability and the speed of execution of random testing.

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. This is also the case for the family of algorithms developed around the work of Chen *et al.* on Adaptive Random Testing (ART) [9]. ART is based on the intuition that an even distribution of test cases in the input space allows finding faults through fewer test cases than with purely random testing. ART generates candidate inputs randomly, and at every step selects from them the one that is furthest away from the already used inputs. ART was initially introduced for numerical values and it calculates the distance between two such values using the Euclidean measure. ART was shown to reduce the number of tests required to reveal the first fault by as much as 50% over purely random testing.

The ideas behind ART are attractive for testing object-oriented (O-O) applications too. The challenge is to define what it means to "spread out" instances of arbitrarily complex types as ART does for numerical values. We have developed a method for automatically calculating an "object distance" [11] and applying it to adaptive random testing of object-oriented applications.

This paper brings some changes to the previously proposed model for the object distance. The new model (presented in Section 2) removes some of the deficiencies of the previous one and is more intuitive. Additionally, this paper makes the following contributions:

- **Implementation**: It presents the implementation of the new testing strategy (called ARTOO) in the AutoTest tool [19] based on the object distance (Sections 3 and 4).
- **Evaluation**: It examines how this strategy performs compared to a directed random testing strategy (Section 5). The

results show that in most cases ARTOO reduces the number of tests required to reveal a fault, and that this difference can be as high as two orders of magnitude.

- **Other applications**: It presents ideas for further applications of the object distance, such as object clustering and integration of manual and automatic testing (Section 6).

2. OBJECT DISTANCE

The object distance is a measure of how different two objects are. It is a general relation that can be applied to any pair of objects in an O-O context. A full definition and examples can be found in our previous publication [11] on this topic. Here we list only the main principles behind the calculation of the object distance and highlight the differences we introduce with respect to the initial model.

In general objects are characterized by:

- their values
- their dynamic types
- recursively the primitive values of the attributes or the objects referred by the attributes.

Hence, the object distance must take into account these three dimensions. Thus, the distance between two composite objects p and q should be a monotonically non-decreasing function of each of the following three components:

- **Elementary distance**: a measure of the difference between the *direct values* of the objects (the values of the references in the case of reference types and the embedded values in the case of primitive types).
- **Type distance**: a measure of the difference between the objects' *types*, completely independent of the values of the objects themselves.
- **Field distance**: a measure of the difference between the objects' individual *fields*[1]. This will be the same notion of object distance, applied recursively. The fields should be compared one by one, considering only "matching" fields corresponding to the same attributes in both objects; non-matching fields also cause a difference, but this difference is captured by the type distance.

Thus, we may express the formula for the distance $p \leftrightarrow q$:

$$\begin{array}{l} p \leftrightarrow q = combination\ (\\ \quad elementary\_distance\ (p,q), \\ \quad type\_distance\ (type\ (p), type\ (q)), \\ \quad field\_distance\ (\{[p.a \leftrightarrow q.a] \\ \qquad |a \in Attributes\ (type\ (p), type\ (q))\})) \end{array} \quad (1)$$

where $Attributes(t1,t2)$ is the set of attributes applicable to both objects of type $t1$ and objects of type $t2$. We look below at possible choices for the functions *combination*, *elementary_distance*, *type_distance*, and *field_distance*.

This formula differs from the one introduced in [11] in that it separates clearly between the notions of elementary distance (which does not involve traversing the object structure) and field distance, which involves the recursive application of the distance calculation to the matching fields of the two objects, regardless of whether the type of these fields is primitive or reference. This unified view is not only more intuitive, but it also removes a deficiency present in the previous model: according to the formula introduced in [11], the distance between two distinct references pointing to the same object was 0. We believe this to be incorrect, since a distance of 0 should indicate identical references. The formula presented here fixes this issue by taking into account the different values of the references in the distance calculation.

We now look at each of the components of the distance.

For the *elementary distance* we define fixed functions for each possible type (primitive or reference) of the compared values p and q:

1. For numbers: $\mathbf{F}(|p-q|)$ (where $\mathbf{F}$ is a monotonically non-decreasing function with $\mathbf{F}(0) = 0$).
2. For characters: 0 if identical, $\mathbf{C}$ otherwise.
3. For booleans: 0 if identical, $\mathbf{B}$ otherwise.
4. For strings: the Levenshtein distance [20].
5. For references: 0 if identical, $\mathbf{R}$ if different but none is void (null), $\mathbf{V}$ if only one of them is void.

In this definition, $\mathbf{C}$, $\mathbf{B}$, $\mathbf{R}$, and $\mathbf{V}$ are positive values chosen conventionally.

The *distance between two types* is a monotonically increasing function of the sum of their path lengths to any closest common ancestor, and of the number of their non-shared features[2]. In languages where all classes have a common ancestor (`ANY` in Eiffel, `Object` in Java), any two classes have a closest common ancestor. If the types of the two compared objects do not have a closest common ancestor, then the object distance is not defined for them, since the objects must always be compared with respect to a common type of which they are direct or indirect instances. Non-shared features are features not inherited from a common ancestor.

We thus obtain the following formula for the type distance between two types t and u:

$$\begin{array}{l} type\_distance\ (t,u) = \\ \quad \lambda * path\_length\ (t,u) + \nu * \sum_{a \in non\_shared\ (t,u)} weight_a \end{array} \quad (2)$$

where *path_length* denotes the minimum path length to a closest common ancestor, and *non_shared* the set of non-shared features. λ and ν are two non-negative constants. $weight_a$ denotes the weight associated with attribute a. It allows for increased flexibility in the distance definition, since thus some attributes can be excluded from the distance (by an associated weight of 0) or can be given increased weight relative to others.

The *field distance* is obtained by recursively applying the distance calculation to all pairs of matching fields of the compared objects:

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

where a iterates over all matching attributes of p and q. We take the arithmetic mean (the sum divided by the number of its elements) to avoid giving too much weight to objects that have large numbers of fields.

The *combination* function is a weighted sum of its three components. The weights (ε for the elementary distance, τ for the type

[1] We use the term *field* as a dynamic notion corresponding to the static notion of *attribute*. In other words, classes have attributes, while objects have fields.

[2] Attributes and methods

distance and α for the field distance) allow for more flexibility in the distance calculation. Furthermore, each of the three components of the distance must be normalized to a bounded interval with the lower limit 0, so that the distances remain comparable. A normalization function $norm(x)$ should be monotonically increasing and fulfill the property $norm(0) = 0$. By convention, we will take 1 as the upper bound of the normalization interval.

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

$$\begin{aligned} p \leftrightarrow q = \tfrac{1}{3} * (& \\ & norm\ (\varepsilon * elementary\_distance\ (p,q)) \\ & + norm\ (\tau * \lambda * path\_length\ (type\ (p), type\ (q)) \\ & + \tau * \nu * \textstyle\sum_{a \in non\_shared\ (type(p), type(q))} weight_a) \\ & + norm\ (\alpha * \overline{\sum_a} weight_a * (p.a \leftrightarrow q.a))) \end{aligned} \tag{4}$$

where in the last term a ranges over all matching fields.

This definition uses several constants and a function, for which any application must choose values. The implementation described in Section 4 uses the following values:

- 1 for all constants except $\alpha = \frac{1}{2}$ and $R = 0.1$
- the normalization function applied to each component of the distance: $(1 - \frac{1}{1+x}) * max\_distance$, where $max\_distance$ is the upper limit of the interval to which the distance must be normalized. As mentioned above, we used a value of 1 for this limit.

While our experience with the model indicates that these simple values suffice, we intend to perform more experiments to determine correlations between the nature of the application under test and the choice of parameters.

3. ADAPTIVE RANDOM TESTING FOR OBJECT-ORIENTED SOFTWARE (ARTOO)

The object distance allows the development of several testing algorithms. We have proposed an algorithm [11] which keeps track of the already used and the available objects and always chooses as input from the available set the object that has the highest average of distances to the already used objects. The algorithm is shown in Figure 1.

This algorithm uses pseudo-code but borrows some notations and conventions from Eiffel. In particular, `ANY` is the root of the class hierarchy: all classes inherit from it by default. The `distance` function is implemented as described in Section 2. For simplicity, the algorithm only computes the sum of the distances and not their average; this is a valid approximation since, to get the average distance for every object in the available set, this sum of distances would have to be divided by the number of objects in the already used set, which is constant at every step of choosing an input.

This algorithm is applied every time a new test input is required. For example, for testing a routine (method) `r` of a class `C` with the signature `r (o1: A; o2: B)`, 3 inputs are necessary: an instance of `C` as the target of the routine call and instances of `A` and `B` as arguments for the call. Hence, ARTOO maintains a list of the objects used for all calls to `r`, and applies the algorithm described above every time a new input is required. In other words, when

```
used_objects: SET [ANY]
candidate_objects: SET [ANY]
current_best_distance: DOUBLE
current_best_object: ANY
v0, v1: ANY
current_accumulation: DOUBLE
...

current_best_distance := 0.0
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_distance)
   then
      current_best_distance := current_accumulation
      current_best_object := v0
   end
end
candidate_objects.remove(current_best_object)
used_objects.add(current_best_object)
run_test(current_best_object)
```

Figure 1: Algorithm for selecting a test input. The object that has the highest average distance to those already used as test inputs is selected.

an instance of `C` is necessary, ARTOO compares all instances of `C` available in the pool of objects to all the instances of `C` already used as targets in test calls to `r`. It selects the one that has the highest average distance to the already used ones, and then repeats the algorithm for picking an instance of `A` to use as first argument in the call, and then does the same for `B`.

This strategy is similar to the one originally proposed for ART [9], the differences being the selection criterion (average distance rather than maximum minimum distance) and the computation of the distance measure.

4. IMPLEMENTATION

We implemented ARTOO as a plug-in for the AutoTest [24] tool. ARTOO is available in open source at `http://se.inf.ethz.ch/people/ciupa/artoo.html`. This section first provides an overview of AutoTest, then describes in detail its algorithm for generating test inputs, then explains how ARTOO is integrated in AutoTest, and finally provides an example that illustrates how ARTOO works.

4.1 AutoTest

AutoTest performs fully automatic unit testing of Eiffel code equipped with contracts. Here we only provide an overview of the tool and present in detail the parts that are particularly relevant for the implementation of ARTOO.

According to the Design by Contract software development methodology [23], contracts (routine pre- and postconditions and class invariants) express elements of the specification of the software. If they are executable, they can be monitored at runtime and any contract violation signals a fault in the executed program. This enables AutoTest to use the contracts present in the code as an automated oracle. AutoTest targets Eiffel code, since Eiffel has embedded support for Design by Contract. AutoTest can also function in the absence of contracts: in such a case, it would report any uncaught exception as a fault.

The released version of AutoTest employs a directed random strategy for input generation (explained in detail in Section 4.2), but the tool has a pluggable architecture so that other strategies for input creation can easily be added. In particular, a command line option selects the desired input generation method out of the available ones. Using the currently selected strategy, AutoTest automatically generates inputs, runs the routine under test with these inputs, and monitors contracts. If it detects any contract violation (except for the case in which a generated test does not fulfill the precondition of the routine under test), it reports a fault. The detected problem may lie either in the implementation or in the contract, requiring further analysis, but this is not significant at this point: regardless of the location of the fault, a contract violation signals a mistake in the developer's thinking, so a testing tool should report it as a fault.

AutoTest uses a two-process model for test execution: a master process knows the testing strategy and gives simple commands (such as object creation, routine invocation, etc.) to a slave process. The slave (an interpreter) is responsible only for executing such instructions and returning responses to the master. This separation of orchestration and execution has the advantage of robustness: if the slave cannot recover from a failure triggered during test execution, the master simply shuts it down and restarts it, resuming testing where it was interrupted.

4.2 Random input generation in AutoTest

AutoTest keeps a pool of objects available for testing; in this pool it stores all objects created as test inputs and returns them once they have been used in tests. The algorithm for input generation proceeds in the following manner, given a routine `r` of a class `C` currently under test. To test `r`, a target object and arguments (if `r` takes any) are necessary. The algorithm either creates new instances for the target object and arguments or uses existing instances from the pool. The decision is taken probabilistically for each required input; for the results presented here we used a probability of 0.25 of creating new objects: a new object is created roughly once every four test case runs. We use this value because previous work [12] determined it to deliver the best results (in terms of the number of found faults) for the random strategy.

If the decision is to create new instances, AutoTest calls a randomly chosen constructor of the corresponding class (or, if the class is abstract, its closest non-abstract descendant). If this constructor takes arguments, the same algorithm is applied recursively. The input generation algorithm treats primitive types (such as `INTEGER`, `REAL`, `CHARACTER`, `BOOLEAN`) differently: for an argument declared of a primitive type, a value is chosen according to a preset probability (also 0.25 as determined by previous experiments) either out of the set of all possible values or a set of predefined special values. These predefined values are assumed to have a high fault-revealing rate when used as inputs; for example, for type `INTEGER`, they include the minimum and maximum possible values, 0, 1, -1, etc. This selection of primitive values from predefined sets makes the input generation not purely random; we hence call it "directed random testing". Please note that this term only refers to the strategy described above and not to homonyms found in the literature ([15], [28]).

To obtain more diverse objects in the pool, the random strategy also performs *diversification* operations: it calls a command (routine that does not return a value and may change the state) on an object selected randomly from the pool. Such a diversification operation occurs with probability 0.5 after every call to a routine under test.

Generating objects by calling constructors and then possibly other routines of the class has the advantage that it produces only valid objects, that is objects that satisfy the class invariant, since it is the job of the constructor to fulfill this class invariant after it is done executing and all subsequently called routines must maintain it.

4.3 ARTOO in AutoTest

We implemented ARTOO as a plug-in strategy for input generation in AutoTest. ARTOO only affects the algorithm used for creating and selecting inputs. The other parts of the testing process (execution in the two processes, using contracts as an oracle) remain in AutoTest as described above, allowing objective performance comparisons between the input generation strategies. This is particularly important if one wants to compare the performance of the two strategies: the conditions under which the experiments are run must be the same.

ARTOO creates new objects and applies diversification operations with the same probabilities as the directed random strategy. It proceeds differently from the latter only with respect to the selection of the objects (composite and primitive) to be used in tests. Its implementation is similar to the algorithm presented in Section 3. The main difference is that, while the latter algorithm does not consider the creation of new objects as it proceeds (in other words, no new objects are added to the set of available inputs), in the implementation new instances are created constantly and then considered for selection as test inputs.

The implementation of ARTOO solves infinite recursion in the field distance by cutting the recursive calculation after a fixed number of steps (2 in the case of the results presented in the next section). Also, the calculation of the object distance is slightly different in the implementation of ARTOO than the formula given in Section 2, in that no normalization is applied to the elementary distances as a whole: for characters, booleans, and reference values the given constants are directly used, and for numbers and strings the normalization function given in Section 2 is applied to the absolute value of the difference (for numbers) and to the Levenshtein distance respectively (for strings). For the field distance, no normalization is necessary, since the averaged distances between the objects referred by the attributes are themselves bounded to the same interval.

4.4 Example

The following example shows how the implementation of ARTOO in AutoTest proceeds. Suppose ARTOO tests the class `BANK_ACCOUNT`, given in Listing 1.

```
class BANK_ACCOUNT
create
  make

feature -- Bank account data
  owner: STRING
  balance: INTEGER

feature -- Initialization
  make (s: STRING; init_bal: INTEGER) is
      -- Create a new bank account.
    require
      positive_initial_balance : init_bal >= 0
      owner_not_void: s /= Void
    do
      owner := s
      balance := init_bal
    ensure
      owner_set: owner = s
      balance_set: balance = init_bal
```

```
    end

feature -- Operation

  withdraw (sum: INTEGER) is ...

  deposit (sum: INTEGER) is ...

  transfer (other_account: BANK_ACCOUNT; sum: INTEGER
      ) is
      -- Transfer 'sum' to 'other_account'.
    require
      can_withdraw: sum <= balance
    do
      balance := balance - sum
      other_account.deposit (sum)
    ensure
      balance_decreased: balance < old balance
      sum_deposited_to_other_account: other_account.balance
          > old other_account.balance
    end
invariant
  owner_not_void: owner /= Void
  positive_balance: balance >= 0
end
```

Listing 1: Part of the code of class BANK_ACCOUNT, which ARTOO must test.

For testing routine `transfer`, ARTOO needs an instance of `BANK_ACCOUNT` as the target of the call, another instance of `BANK_ACCOUNT` as the first argument, and an integer as the second argument. For the first test call to this routine, there are no inputs previously used, so ARTOO will pick objects with corresponding types at random from the pool. Suppose at this point the pool contains the following objects:

ba1: *BANK_ACCOUNT*, *ba1.owner*=**"A"**, *ba1.balance*=675234
ba2: *BANK_ACCOUNT*, *ba2.owner*=**"B"**, *ba2.balance*=10
ba3: *BANK_ACCOUNT*, *ba3.owner*=**"O"**, *ba3.balance*=99
ba4 = *Void*
i1: *INTEGER*, *i1* = 100
i2: *INTEGER*, *i2* = 284749
i3: *INTEGER*, *i3* = 0
i4: *INTEGER*, *i4* = −36452
i5: *INTEGER*, *i5* = 1

Suppose ARTOO picks `ba3` as target, `ba1` as first argument and `i5` as second argument for the first call to `transfer`. These 3 values are saved to disk [3] and then the call is executed:

ba3. *transfer* (*ba1*, *i5*)

The state of the object pool is now as follows:

ba1: *BANK_ACCOUNT*, *ba1.owner*=**"A"**, *ba1.balance*=675235
ba2: *BANK_ACCOUNT*, *ba2.owner*=**"B"**, *ba2.balance*=10
ba3: *BANK_ACCOUNT*, *ba3.owner*=**"O"**, *ba3.balance*=98
ba4 = *Void*
i1: *INTEGER*, *i1* = 100
i2: *INTEGER*, *i2* = 284749
i3: *INTEGER*, *i3* = 0
i4: *INTEGER*, *i4* = −36452
i5: *INTEGER*, *i5* = 1

For the next call to `transfer`, ARTOO chooses a target by picking the non-Void object of type `BANK_ACCOUNT` that is furthest from the already used target. For the first argument it picks the instance of `BANK_ACCOUNT` that is furthest from the first argument previously used, and likewise for the integer argument. It thus chooses `ba1` as target, `ba4` as first argument (Void references always have the maximum possible distance to non-void references), and `i2` as second argument. These values are saved and the call is executed:

ba1. *transfer* (*ba4*, *i2*)

This triggers an attempt to call a routine on a Void target (through the instruction `other_account.deposit (sum)` in the body of routine `transfer`), which results in an exception, so ARTOO has found a fault in routine `transfer`. The precondition of this routine should state that `other_account` must be non-Void.

The pool is not changed and ARTOO picks again the objects with the highest average distances to the already used ones. So `ba2` is chosen as both target and first argument, and `i4` as second argument. The values are saved and the call is executed:

ba2. *transfer* (*ba2*, *i4*)

This triggers a postcondition violation in `transfer` since trying to transfer a negative amount from the current account does not reduce the balance of the current account. ARTOO has thus found another fault, since transferring negative amounts should not be allowed. It is interesting to note that this call actually exposes another fault, namely that transferring money from an account to itself should not be allowed.

For simplicity this example did not consider the creation of objects between test calls. In practice, objects are created with a certain probability and added to the pool between calls to routines under test, so ARTOO also considers these new instances when selecting the inputs.

5. EXPERIMENTAL RESULTS

5.1 Experimental setup

It is important to evaluate testing tools on real-world code, which can be tested as it is and was not written only for the purposes of a specific experiment or tool evaluation study. Practical applicability of the tool is one of the major conditions required for its acceptance by the testing community.

The subjects used in the evaluation of ARTOO are classes from the EiffelBase library [2] version 5.6. EiffelBase is an industrial-grade library used by virtually all projects written in ISE Eiffel, similar to the System library in Java or C#. No changes were made to this library for the purposes of this experiment: the classes tested are taken from the released, publicly-available version of the library and all faults mentioned are real faults, present in the 5.6 release of the library.

Table 1 presents some properties of the classes under test. All the metrics except the last column refer to the flat form of the classes,

[3] It is necessary to save them before the call to the routine because the execution of the routine might change their state or it might crash the process in which it is executing.

that is a form of the class text that includes all the features of the class at the same level, regardless of whether they are inherited or introduced in the class itself.

All tests were run using the ISE Eiffel compiler version 5.6 on a machine having a Pentium M 2.13 GHz processor, 2 GB of RAM, and running Windows XP SP2. The tests applied both the directed random strategy (called RAND for brevity below) and ARTOO, testing one class at a time. Since the seed of the pseudo-random number generator influences the results, the results presented below are averaged out over 5 10-minute tests of each class using different seeds.

It is important to note that the testing strategy against which we compare ARTOO is in fact not purely random, as explained in Section 4.2: values for primitive types are not always selected randomly from the set of all possible values, but from a restricted, predefined set of values considered to be more likely to uncover faults. We chose this strategy as the basis for comparison because previous experiments [12] have shown it to be more efficient than purely random testing.

The results were evaluated according to two factors: *number of tests to first fault* and *time to first fault*. Other criteria for the evaluation (such as various measures of code coverage) are also possible. However, we consider the fault-detecting ability of a testing strategy to be its most important property. Measuring the time elapsed and the number of test cases run until the first fault is found is driven by practical considerations: software projects in industry usually run under tight schedules, so the efficiency of any testing tool plays a key role in its success.

5.2 Results

Table 2 shows a routine-by-routine comparison of the time to first fault and tests to first fault for both ARTOO and RAND applied to classes `ARRAYED_LIST` and `ACTION_SEQUENCE`. The table shows, for each routine in which both strategies found faults, the number of tests and time required by each strategy to find a fault in that particular routine. Both the tests and the time are averages, on the five seeds, of the time elapsed since the beginning of the testing session and the decimal part is omitted. All calls to routines and creation procedures of the class under test are counted as test cases for that class. The table also shows, for each of these two factors, the ratios between the performance of ARTOO and that of RAND rounded to two decimal digits, showing in bold the cases for which ARTOO performed better.

At courser granularity, Table 3 shows for every class under test the average (over all routines where both strategies found at least one fault) of the number of tests to first fault and time to first fault for each strategy and the proportions ARTOO/RAND, rounded to two decimal digits. Figures 2 and 3 show the same information, comparing for every class the number of tests to first fault and the time to first fault, respectively.

In most cases ARTOO reduces the number of tests necessary to find a fault by a considerable amount, sometimes even by two orders of magnitude. However, calculating the object distances is time-consuming. The overhead ARTOO introduces for selecting which objects to use in tests (the distance calculations, the serializations of objects, etc.) causes it to run fewer tests over the same time than RAND. For the tested classes, which all have fast-executing routines, although ARTOO needs to run fewer tests to find faults, RAND needs less time.

The experiments also show that there are cases in which ARTOO finds faults which RAND does not find (over the same test duration). Table 4 lists the classes and routines where only ARTOO was able to find some faults, the number of tests and the time

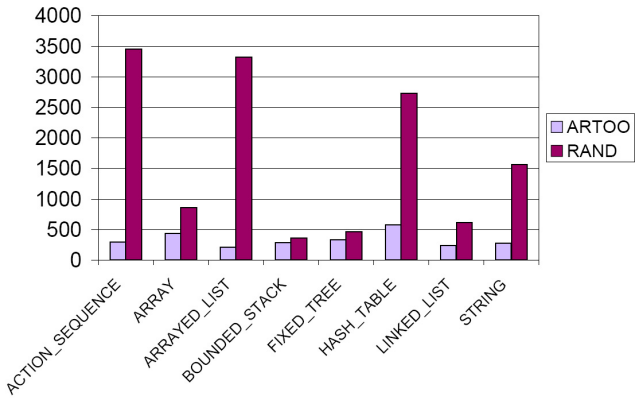

Figure 2: Comparison of the average number of tests cases to first fault required by the two strategies for every class. ARTOO constantly outperforms RAND.

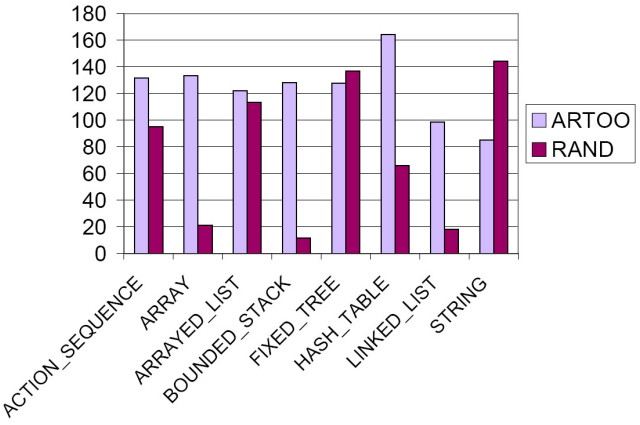

Figure 3: Comparison of the average time to first fault required by the two strategies for every class. RAND is generally better than ARTOO.

that ARTOO needed to find the first fault, and the number of faults it found in each routine.

Class	Routine	Tests to first fault	Time to first fault (seconds)	#faults
ARRAYED_LIST	remove	167	46	1
FIXED_TREE	child_is_last	717	283	1
FIXED_TREE	duplicate	422	134	1
STRING	grow	492	163	2
STRING	multiply	76	17	2

Table 4: Faults which only ARTOO finds

Class	Total lines of code	Lines of contract code	#Routines	#Attributes	#Parent classes
ACTION_SEQUENCE	2477	164	156	16	24
ARRAY	1208	98	86	4	11
ARRAYED_LIST	2164	146	39	6	23
BOUNDED_STACK	779	56	62	4	10
FIXED_TREE	1623	82	125	6	4
HASH_TABLE	1791	178	122	13	9
LINKED_LIST	1893	92	106	6	19
STRING	2980	296	171	4	16

Table 1: Properties of the classes under test

Class	Routine	Tests to first fault			Time to first fault (seconds)		
		ARTOO	RAND	$\frac{ARTOO}{RAND}$	ARTOO	RAND	$\frac{ARTOO}{RAND}$
ARRAYED_LIST	append	**432**	**5517**	**0.08**	311	191	1.62
	do_all	**296**	**737**	**0.40**	137	18	7.48
	do_if	**16**	**1258**	**0.01**	**2**	**39**	**0.05**
	fill	**159**	**7130**	**0.02**	**40**	**256**	**0.16**
	for_all	**303**	**517**	**0.59**	138	17	7.93
	is_inserted	**31**	**126**	**0.25**	**3**	**7**	**0.43**
	make	23	3	7.44	2	1	2.80
	make_filled	**13**	**117**	**0.11**	**2**	**4**	**0.50**
	prune_all	**51**	**10798**	**0.00**	**3**	**367**	**0.01**
	put	96	89	1.08	11	4	2.67
	put_left	**146**	**9739**	**0.01**	**32**	**331**	**0.10**
	put_right	**278**	**8222**	**0.03**	**132**	**291**	**0.45**
	resize	**355**	**1143**	**0.31**	320	30	10.40
	there_exists	**307**	**518**	**0.59**	151	17	8.78
	wipe_out	**594**	**3848**	**0.15**	546	123	4.41
ACTION_SEQUENCE	arrayed_list_make	**748**	**6800**	**0.11**	564	174	3.24
	call	**109**	**2382**	**0.05**	**10**	**67**	**0.15**
	duplicate	**378**	**410**	**0.92**	196	13	14.46
	for_all	**286**	**623**	**0.46**	64	21	3.00
	is_inserted	115	95	1.21	5	2	2.36
	make_filled	**183**	**449**	**0.41**	49	13	3.65
	put	81	67	1.21	4	4	1.15
	remove_right	**448**	**17892**	**0.03**	**201**	**475**	**0.42**
	resize	**399**	**5351**	**0.07**	187	160	1.17
	set_source_connection_agent	**265**	**3771**	**0.07**	**96**	**112**	**0.86**
	there_exists	215	104	2.07	67	2	33.83

Table 2: Results for two of the tested classes, showing the time and number of tests required by ARTOO and RAND to uncover the first fault in each routine in which they both found at least one fault, and their relative performance. In most cases ARTOO requires significantly less tests to find a fault, but entails a time overhead.

It is also true that RAND finds faults which ARTOO does not find in the same time. This suggests that the two strategies have different strengths and, in a fully automated testing process, should ideally be used in combination. Previous case studies [12] indicate that the evolution of the number of new faults that RAND finds is inversely proportional to the elapsed time. This means in particular that after running random tests for a certain time, it becomes unlikely to uncover new faults. At this point ARTOO can be used to uncover any remaining faults. In cases where the execution time of the routines under test is high, ARTOO is more attractive due to the reduced number of tests it generates before it uncovers a fault.

The results also show that ARTOO is generally less sensitive than RAND to the variation of the seed value, so that its performance is more predictable. Table 5 shows this in terms of the standard deviation of the number of faults found by each strategy for each class under test after 1, 2, 5, and 10 minutes of testing, and of the average and standard deviation of the standard deviations for each strategy. The results are however not indicative of any clear relation between the testing timeout and the variation with the seed of the number of found faults.

5.3 Discussion

We chose to compare the performance of the two strategies when run over the same duration, although other similar comparative studies in the literature use rather the number of generated tests or an achieved level of code coverage as the stopping criterion. We preferred to use time because, in industrial projects and especially in the testing phases of these projects, time is probably the most constraining factor.

These results show that, compared to a directed random testing

Class	Tests to first fault			Time to first fault (seconds)		
	ARTOO	RAND	ARTOO/RAND	ARTOO	RAND	ARTOO/RAND
ACTION_SEQUENCE	**293.72**	**3449.76**	**0.09**	131.53	95.11	1.38
ARRAY	**437.19**	**856.39**	**0.51**	133.21	21.23	6.27
ARRAYED_LIST	**206.80**	**3317.80**	**0.06**	122.16	113.42	1.07
BOUNDED_STACK	**282.50**	**357.17**	**0.79**	128.00	11.45	11.18
FIXED_TREE	**333.99**	**463.91**	**0.71**	**127.73**	**136.64**	**0.93**
HASH_TABLE	**581.21**	**2734.42**	**0.21**	164.41	65.85	2.49
LINKED_LIST	**238.20**	**616.71**	**0.38**	98.39	18.14	5.42
STRING	**279.64**	**1561.60**	**0.17**	**85.03**	**144.28**	**0.58**
Overall averages	*331.66*	*1669.72*	*0.19*	*123.81*	*75.77*	*1.63*

Table 3: Averaged results per class. ARTOO constantly requires fewer tests to find the first fault: on average 5 times less tests than RAND. The overhead that the distance calculations introduce in the testing process causes ARTOO to require on average 1.6 times more time than RAND to find the first fault.

strategy, ARTOO generally reduces the number of tests required until a fault is found, but suffers from a time performance penalty due to the extra computations required for calculating the object distances. The times reported here are *total* testing times; they include both the time spent on generating and selecting test cases and the time spent on actually running the tests. Total time is the measure that most resembles how the testing tool would be used in practice, but this measure is highly dependent on the time spent running the software under test. The test scope of the experiment described above consists of library classes whose routines generally implement relatively simple computations and average at below 20 LOC/routine. When testing more computation-intensive applications, the number of tests that can be run per time unit naturally decreases, hence the testing strategy that needs less tests to uncover faults would be favored.

The biggest threat to the validity of these results is probably the test scope: the limited number of seeds and the tested classes. The results presented here were obtained by averaging out over 5 seeds of the pseudo-random number generator. Given the role randomness plays in both the compared testing algorithms, averaging out over more seeds would produce more reliable results. Likewise, we have chosen the tested classes so that they are fairly diverse (in terms of their semantics and of various code metrics), but testing more classes would yield more generalizable results. We intend to extend the scope of the case study in both these directions.

6. OPTIMIZATIONS

Having the possibility to compute a distance between objects allows clustering techniques to be applied to objects. Any of the known clustering algorithms can be applied based on this distance, so it is possible to group together objects that are similar enough. This allows ARTOO to be optimized by computing the distances to the cluster centers only, and not to each individual object. The precision is dependent on the maximum distance between two distinct objects within a given cluster. A preliminary implementation of this testing strategy shows an average improvement of the time to first fault over ARTOO of 25% at no cost in terms of faults found.

Another improvement is to use information contained in manual tests to further guide the search for fault-revealing inputs. The assumption is that manually written tests for a certain class have inputs more likely to reveal faults than random ones. The generation of inputs is still random, as described above; the information from the manual tests influences only the process of selecting a value to use as input out of the available ones, very much like the basic version of ARTOO does. This strategy uses two measures to assess the desirability of using a certain value as test input: how close this value is to the manual inputs and how far it is from already used automatic inputs. Preliminary experiments using a first prototype implementation of this testing strategy show that it can reduce the number of tests to first fault by an average factor of 2 compared to the basic implementation of ARTOO described above.

The implementation of ARTOO used in the experiments uses a "complete" definition of the object distance, as described in Section 2. A less computationally intensive definition of the object distance might still require fewer tests to uncover a fault, but also less time. We intend to also explore such alternatives in order to improve the performance of ARTOO.

The definition of the object distance provided in Section 2 only uses the syntactic form of the objects and does not take their semantics into account in any way. This approach has the merit of being completely automatable. Integrating semantics into the computation would require human intervention, but would certainly enrich the model and its accuracy and allow finer-grained control of the developer over the testing process in ARTOO. Some support for this is already available through the constants used in the object distance calculation, whose values can easily be changed for fine tuning the distance. In our future work we intend to investigate this idea and other possibilities for providing further support for integrating semantics into the object distance.

7. RELATED WORK

Random testing presents several benefits in automated testing processes: ease of implementation, efficiency of test case generation, and the existence of ways to estimate its reliability [17]. Many reference texts are, however, critical of it. Glenford J. Myers deems it the poorest testing methodology and "at best, an inefficient and ad hoc approach to testing" [25].

Several studies [14, 16] disproved this assessment by showing that random testing can be more cost-effective than partition testing. Andrews *et al.* [3] show that, when specific recommended practices are followed, a testing strategy based on random input generation finds faults 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.

Although random testing of numerical applications has a longer standing tradition than random testing of object-oriented software, the interest for the latter has increased in recent years. Tools like JCrasher [13], Eclat [27], Jtest [1], Jartege [26], or RUTE-J [4] are proof of this interest. All these tools employ purely random

Class	Timeout (minutes)	StDev(NumberFoundFaults) ARTOO	RAND
ACTION_SEQUENCE	1	1.87	2.92
	2	1.14	2.59
	5	0.89	1.22
	10	1.64	0.71
ARRAY	1	2.30	13.22
	2	2.45	16.81
	5	2.77	17.04
	10	5.27	17.04
ARRAYED_LIST	1	2.95	1.52
	2	3.08	3.51
	5	3.81	8.37
	10	4.93	12.60
BOUNDED_STACK	1	2.35	1.10
	2	3.56	0.84
	5	3.11	1.22
	10	2.17	1.48
FIXED_TREE	1	2.30	3.91
	2	1.30	2.70
	5	1.64	2.70
	10	2.59	2.17
HASH_TABLE	1	0.89	2.12
	2	1.64	2.05
	5	2.05	5.15
	10	3.11	7.91
LINKED_LIST	1	0.55	1.48
	2	0.45	1.79
	5	1.34	2.17
	10	1.14	4.22
STRING	1	2.07	1.14
	2	3.13	0.44
	5	3.7	1
	10	3.91	0.83
Average		*2.37*	*4.49*
Standard deviation		*1.19*	*5.14*

Table 5: Standard deviations of numbers of found faults for each strategy due to the influence of the seed for the pseudo-random number generator, and the average and standard deviation of the standard deviations for each strategy. ARTOO is generally less sensitive to the choice of seed.

strategies for input generation (possibly in combination with also allowing users to define inputs [26]), while ARTOO's input selection criterion is based on a measure of how different the candidate objects are from the ones already used.

Other research based on the idea of random testing tries to improve its performance by adding some guidance to the algorithm. Such guidance can mean pruning out invalid and duplicate inputs [28], combining random and systematic techniques [15], or trying to spread out the selected values over the input domain, as is the case for Adaptive Random Testing [9] and quasi-random testing [10]. Based on the ART intuition, a series of related algorithms have been proposed. Mirror ART [8] and ART through dynamic partitioning [7] reduce the overhead of ART. Restricted Random Testing [6] (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 lattice-based ART [22] and ART by bisection with restriction [21].

8. CONCLUSIONS

We have presented an implementation (Sections 3 and 4) and experimental results (Section 5) evaluating the efficiency of a testing strategy called ARTOO. This strategy is based on the ideas of Adaptive Random Testing (ART): it selects test inputs that are the furthest apart in a set of randomly generated values. Adaptive Random Testing was introduced for and could only be applied to numeric values, for which the distance calculation is straightforward. ARTOO extends the applicability of the basic ART algorithm to O-O software by using the object distance [11] — a measure of how different two objects are. This measure takes into account an elementary distance between the direct values of the objects, a distance between the types of the objects, and recursive distances between the fields of the objects.

The experimental results show that ARTOO finds real faults in real software. Compared to directed random testing, ARTOO significantly reduces the number of tests generated and run until the first fault is found, on average by a factor of 5 and sometimes by as much as two orders of magnitude. The guided input selection process that ARTOO employs does, however, entail an overhead which is not present in unguided random testing. This overhead leads to ARTOO being on average 1.6 times slower than directed random testing in finding the first fault. These results indicate that ARTOO, at least in its current implementation, should be applied in settings where the *number* of test cases generated until a fault is found is more important than the time it takes to generate these test cases — in other words, settings in which the cost of running and evaluating test cases is higher than the cost of generating them. This can be the case, for instance, when there is no automated oracle and thus manual inspection of test cases is necessary. As discussed in Section 6, we are currently investigating ways of improving ARTOO's performance, such as reducing the precision and thus the overhead of the distance calculation or employing object clustering techniques that would allow to calculate distances between clusters of objects rather than between all pairs of objects.

The results also show that ARTOO finds faults that directed random testing does not find in the same time and that ARTOO is less sensitive than directed random testing to the influence of the random part of the algorithm for input generation: the seed for the pseudo-random number generator influences the results less in ARTOO than in directed random testing.

ARTOO is available in open source at `http://se.inf.ethz.ch/people/ciupa/artoo.html`. It is provided as a plug-in to the AutoTest framework.

Future work includes, as mentioned, improving ARTOO's performance and examining how changes to the object distance calculation would affect ARTOO's fault finding ability; in particular, we plan to develop a model for the object distance computation which integrates object semantics based on information provided by the developer. We are also looking into applications other than testing for the object distance.

9. REFERENCES

[1] Jtest. Parasoft Corporation. http://www.parasoft.com/.

[2] The EiffelBase Library. Eiffel Software Inc. http://www.eiffel.com/.

[3] ANDREWS, J. H., HALDAR, S., LEI, Y., AND LI, C. H. Randomized unit testing: Tool support and best practices. Tech. Rep. 663, Department of Computer Science, University of Western Ontario, January 2006.

[4] ANDREWS, J. H., HALDAR, S., LEI, Y., AND LI, F. C. H. Tool support for randomized unit testing. In *RT '06:*

Proceedings of the 1st International Workshop on Random Testing (2006), ACM Press, New York, NY, USA, pp. 36–45.

[5] BOYAPATI, C., KHURSHID, S., AND MARINOV, D. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy* (2002).

[6] CHAN, K. P., CHEN, T. Y., AND TOWEY, D. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality* (2002), Springer-Verlag, London, UK, pp. 321 – 330.

[7] CHEN, T., MERKEL, R., WONG, P., AND EDDY, G. Adaptive random testing through dynamic partitioning. In *Proceedings of the Fourth International Conference on Quality Software* (Los Alamitos, CA, USA, 2004), IEEE Computer Society, pp. 79 – 86.

[8] CHEN, T. Y., KUO, F. C., MERKEL, R. G., AND NG, S. P. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software* (Los Alamitos, CA, USA, 2003), IEEE Computer Society, pp. 4 – 11.

[9] CHEN, T. Y., LEUNG, H., AND MAK, I. K. Adaptive random testing. In *Advances in Computer Science - ASIAN 2004: Higher-Level Decision Making. 9th Asian Computing Science Conference. Proceedings* (2004), M. J. Maher, Ed., Springer-Verlag GmbH.

[10] CHEN, T. Y., AND MERKEL, R. Quasi-random testing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (2005), ACM Press, New York, NY, USA, pp. 309–312.

[11] CIUPA, I., LEITNER, A., ORIOL, M., AND MEYER, B. Object distance and its application to adaptive random testing of object-oriented programs. In *RT '06: Proceedings of the 1st International Workshop on Random Testing* (2006), ACM Press, New York, NY, USA, pp. 55–63.

[12] CIUPA, I., LEITNER, A., ORIOL, M., AND MEYER, B. Experimental assessment of random testing for object-oriented software. In *Proceedings of ISSTA'07: International Symposium on Software Testing and Analysis 2007* (2007), ACM, New York, NY, USA, pp. 84 – 94.

[13] CSALLNER, C., AND SMARAGDAKIS, Y. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience 34*, 11 (2004), 1025–1050.

[14] DURAN, J., AND NTAFOS, S. An evaluation of random testing. *IEEE Transactions on Software Engineering SE-10* (July 1984), 438 – 444.

[15] GODEFROID, P., KLARLUND, N., AND SEN, K. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), ACM Press, New York, NY, USA, pp. 213–223.

[16] HAMLET, D., AND TAYLOR, R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering 16 (12)* (December 1990), 1402–1411.

[17] HAMLET, R. Random testing. In *Encyclopedia of Software Engineering*, J. Marciniak, Ed. Wiley, 1994, pp. 970–978.

[18] KHURSHID, S., PASAREANU, C. S., AND VISSER, W. 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)* (2003), vol. LNCS 2619, Springer-Verlag, pp. 553–568.

[19] LEITNER, A., AND CIUPA, I. AutoTest. http://se.inf.ethz.ch/people/leitner/auto_test/, 2005 - 2007.

[20] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Doklady Akademii Nauk SSSR 163*, 4 (1965), 845–848.

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

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

[23] MEYER, B. *Object-Oriented Software Construction, 2nd edition*. Prentice Hall, 1997.

[24] MEYER, B., CIUPA, I., LEITNER, A., AND LIU, L. L. Automatic testing of object-oriented software. In *Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science)* (2007), J. van Leeuwen, Ed., Lecture Notes in Computer Science, Springer-Verlag.

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

[26] ORIAT, C. Jartege: a tool for random generation of unit tests for Java classes. Tech. Rep. RR-1069-I, Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universite Joseph Fourier Grenoble I, June 2004.

[27] PACHECO, C., AND ERNST, M. D. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference* (Glasgow, Scotland, July 25–29, 2005).

[28] PACHECO, C., LAHIRI, S. K., ERNST, M. D., AND BALL, T. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering* (2007).

[29] TONELLA, P. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), ACM Press, New York, NY, USA, pp. 119–128.

[30] VISSER, W., PASAREANU, C. S., AND KHURSHID, S. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (2004), ACM Press, New York, NY, USA, pp. 97–107.