On the Predictability of Random Tests for Object-Oriented Software

Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, Bertrand Meyer
Department of Computer Science, ETH Zurich, Switzerland
{ilinca.ciupa, alexander.pretschner, andreas.leitner, manuel.oriol, bertrand.meyer}@inf.ethz.ch

Abstract

Intuition suggests that random testing of object-oriented programs should exhibit a significant difference in the number of faults detected by two different runs of equal duration. As a consequence, random testing would be rather unpredictable. We evaluate the variance of the number of faults detected by random testing over time. We present the results of an empirical study that is based on 1215 hours of randomly testing 27 Eiffel classes, each with 30 seeds of the random number generator. Analyzing over 6 million failures triggered during the experiments, the study provides evidence that the relative number *of faults detected by random testing over time is predictable but that different runs of the random test case generator detect* different faults. *The study also shows that random testing quickly finds faults: the first failure is likely to be triggered within 30 seconds.*

1 Introduction

The random generation of test input data is attractive because it is widely applicable and cheap, both in terms of implementation effort and execution time. Yet, in addition to the input data, *test cases* also contain an expected output part. Because it depends on a specific input, the expected output cannot be generated at random. However, it can be provided at different levels of abstraction [27]. One extreme possibility is to specify the expected output as abstractly as "no exception is thrown." In an admittedly rough manner, this solves the oracle problem: random test case generation becomes as simple as picking elements from the input domain and adding "no exception" as expected output.

Testing object-oriented programs is far more challenging, because the input domain may consist of arbitrarily complex objects: picking random elements from the set of integers is obviously simpler than generating arbitrary electronic health records. Most methods defined for a health record are likely to be applicable only if the health record exhibits certain characteristics—for instance, a comparison of two diagnoses at least requires the existence of the two diagnoses. As a consequence, generating objects to use as input to a method is a non-trivial task.

A particular class of object-oriented software is that of Eiffel programs. One distinctive feature of Eiffel programs is the existence of contracts. Among other things, contracts contain method postconditions. The latter naturally lend themselves to be used as oracles; the respective level of abstraction is somewhere in-between the concrete output and the abstract absence of exceptions. Randomly generating test cases for Eiffel programs hence consists of (1) generating input objects for a method to be tested and (2) adding the postcondition as the expected output.

Previous experiments on the effectiveness of random tests for Eiffel programs [7] produced preliminary results on the number of detected faults over time and on several technical parameters that influence the generation process. These experiments exhibited a seemingly high variation of the number of faults detected over time. From an engineer's point of view, a high variance means low predictability of the process – which immediately reduces its value. One might argue that random testing can be performed overnight and when spare processor cycles are available; the sheer amount of continuous testing would then compensate for any potential variance. However, arbitrary computation resources may not be available, and insights into the efficiency of a testing strategy are useful from the management perspective: such numbers make it comparable to other strategies.

Problem We set out to answer the following questions. How predictable is random testing? What are the consequences? In particular, how would this technique be best used? Should testers constantly run it in the background?

Solution Using our previously developed test case generator, AutoTest [20], we generate and run random tests for 27 classes from a widely used Eiffel library. Each class is tested for 90 minutes. To assess the predictability of the process, we repeat the testing process for each class 30 times with different seeds for the pseudo-random number generator. This results in an overall testing time of 1215 hours with

more than 6 million triggered failures. The main results are the following.

1. When averaging over all 27 classes, 30% of the overall number of faults detected during the experiments are found after 10 minutes. After 90 minutes, on average, an additional 8 percent points of the overall number of randomly detected faults are found.

2. In terms of the relative number of detected faults (relative to the overall number of faults detected via random testing), random testing is highly predictable, as measured by a low standard deviation.

3. Different runs of the testing process reveal different faults.

4. For each of the classes, at least one out of the 30 experiments detected a fault within the first second. This can be read as follows: random testing detects a fault within at most 30 seconds.

A package including the results and the source code of AutoTest is available online.[1] It contains everything needed for the replication and extension of our experiments.

Contributions This paper presents empirical evidence on the predictability of random testing. While the effectiveness of random testing has been studied before, we do not know of any other investigations of the *predictability* of the process.

Overview The remainder of this paper is organized as follows. We describe our conceptual framework for randomly testing OO programs in §2 and justify the choice of parameters measured in the experiments. Technological matters and a comprehensive description of the experiments are the subject of §3. We state observations and analyze them in §4. Our work is put in context in §5, and we conclude in §6.

2 Random Tests for OO Programs

This section first defines the notions of test case, fault and failure as used throughout this paper. It then presents the test case generation algorithm and explains it on an example test case. The section closes with an explanation of our reasons for choosing time (rather than number of generated and executed test cases) as a stopping criterion for testing.

[1] `http://se.inf.ethz.ch/people/ciupa/public/random_oo_testing_experiment.zip`

2.1 Unit Tests

Intuitively, the purpose of unit testing is to verify the run of a certain program unit. Unit tests for OO programs must take into consideration:

Sub-method invocations – While the purpose of a unit test (xUnit-style) is most often to verify the run of a particular method, a unit test typically does not only cause the invocation of the method under test. This is because the method under test may itself call other methods. In a strict sense, the unit under test is the originally called method only. Methods called by the method under test are not part of the unit under test. From a more pragmatic point of view some of the methods invoked by the method under test may belong to the unit under test. It is not clear for which methods this applies, though.

Inputs – Input data can be created in several ways: by treating state as a bit field and assembling it bit by bit, by creating the required input objects via constructor invocations, by providing mock objects with predefined behaviors, etc.

Oracle – The oracle of a test case can be provided at different levels of abstraction: it can be as concrete as specifying exactly the expected output, it can specify a condition that the output should fulfill for a particular input, it can specify conditions that the output should fulfill for any input, or it can be as abstract as specifying "no exception" as expected output.

The experiments described in this paper ran for 1215 hours. For experiments of this size it is important that *the notion of test case is chosen with automation and efficiency in mind*. Input objects are created through regular constructor calls, because it is much more likely to end up with a valid object (one that satisfies its invariant) when creating it through a constructor call than by setting arbitrary bits in a bit-vector. Hence creation through execution promises to be more efficient. For the same reason, objects created for testing a certain method are kept in a pool and can be reused later for further testing. This might then change the objects' state and will also return them to the pool after the method call is done executing.

In order to achieve full automation the oracle is solely a contract-based one. The system under test is written in Eiffel, which directly supports contracts in the form of method pre- and postconditions and class invariants. Eiffel programs, including industrial ones, do contain these assertions [5]. Contract violations raise exceptions – the foundation of the contract-based oracle. If an exception is triggered by the method under test or any method that it transitively calls, a failure has been triggered in the system under test. Precondition violations triggered by the test driver calling the method under test outside of its intended use (in other words, violating its precondition) are naturally not classified

```
1 create {STRING} v1.make_empty
2 create {BANK_ACCOUNT} v2.make (v1)
3 v3 := 452719
4 v2.deposit (v3)
5 v4 := Void
6 v2.transfer (v4, v3)
...
```

Figure 1. Example test case generated by AutoTest. Methods deposit and transfer of class BANK_ACCOUNT are being tested.

as failures. The AutoTest framework classifies test cases into the following categories: passed (no exception), unresolved (precondition violation in method under test), or failed (other exception).

2.2 Faults and Failures

In this paper, the notions of faults and failures are largely determined by the contract-based oracle. In general, a *failure* is an observed difference between actual and intended behaviors. In this paper, we interpret every contract violation (except for immediate precondition violations) as a failure. However, programmers are interested in *faults* in the software: wrong pieces of code that trigger the failures, and the same fault may trigger arbitrarily many failures. Hence, an analysis of random testing should consider the detected faults, not the failures. Mapping failures to faults is part of the debugging process and is usually done by humans. For practical reasons, this is not feasible for the over 6 million failures triggered in this experiment. Instead we rely on an approximation that groups failures based on the following assumption: two failures are a consequence of the same fault if and only if they manifest themselves through the same type of exception, being thrown from the same method and the same class. Throughout this paper, we use the term "fault" in this sense.

2.3 Test Case Synthesis Algorithm

Figures 2 and 3 describe the test case generation algorithm used for the experiments in this paper. This algorithm has been implemented for the AutoTest framework, which was used to execute the tests. The following describes how the generation algorithm produces a test case via an example, depicted in Figure 1.

The language used to express test cases is a simplified, dynamically typed variant of Eiffel. In the example, variables *v1*, *v2*, *v3*, and *v4* represent the object pool.

The example assumes that a class *BANK_ACCOUNT* is being tested. This means that all its methods must be tested. At every step, method *write_tests* (shown in Figure 2) of AutoTest chooses one of the methods which have been tested the least up to that point. When the testing session starts, no methods have been tested, so one of the methods of class *BANK_ACCOUNT* is chosen at random. Assume method *deposit* is chosen. In order to execute this method, one needs an object of type *BANK_ACCOUNT* and an integer representing the amount of money to deposit.

The creation and selection of these values takes place in method *write_test_for_method* , shown in Figure 3. The test generator randomly chooses inputs with the required types from the object pool. However, before it makes this choice it might also create new instances for the required types (with probability $P(gen\_new)$) and add them to the pool. In the example test case the generator decides at this point to create a new object of type *BANK_ACCOUNT*. Therefore it chooses a constructor (*make*) and now works on the sub-task of acquiring objects serving as parameters for this constructor. The constructor *make* requires only one argument which is of type *STRING*. Hence *write_creation* calls itself recursively now with the task of creating a string object. The type of string objects in Eiffel is a (regular) reference type. The algorithm decides again to create a new object, and uses the constructor *make_empty* which does not take any arguments (line 1). The object pool now is: $\{v1 : STRING\}$. The recursive call of *write_creation* returns. Method *write_creation* itself synthesizes the creation instruction for the bank account object (line 2) using the newly created string object. This updates the object pool to: $\{v1 : STRING, v2 : BANK\_ACCOUNT\}$. At this point, *write_creation* terminates, having created a target object. However, it is invoked once more in order to find an integer (the argument to *deposit*). Integers are basic objects and in the example, a random integer (452719) is chosen and assigned to a fresh pool variable (line 3). This changes the object pool to $\{v1 : STRING, v2 : BANK\_ACCOUNT, v3 = 452719\}$. Execution returns to *write_test_for_method* which is now able to synthesize what it was asked for: a call to *BANK_ACCOUNT.deposit*. It uses the newly created bank account and the randomly chosen integer (line 4).

At this point, execution returns to *write_tests* which selects another method for testing. Assume *BANK_ACCOUNT.transfer* is chosen. This method transfers an amount from the bank account object on which it is called to the bank account that is provided as argument. One target object and two arguments are necessary. For each of these three inputs, an object of the corresponding type might be created and added to the pool (with the probability $P(gen\_new)$). In the case of the call to *BANK_ACCOUNT .transfer*, the decision is to create a new bank account. Whenever AutoTest has to create a new object, it may also choose to add Void to the pool instead of a new object. This happens in the example, so the pool now consists of

$\{v1 : STRING, v2 : BANK\_ACCOUNT, v3 = 452719, v4 = Void\}$. Now, two instances of *BANK_ACCOUNT* and an integer are chosen randomly from the pool and the result is the call to *transfer* as shown on line 6. Test case generation continues after this point, but for brevity the example stops here.

The input generation algorithm uses two parameters: the probability of generating new objects at every step $P(gen\_new)$ and the probability of picking values for basic types from all possible values for that type rather than from a predefined set $P(gen\_basic\_rand)$. The experiments described here used $P(gen\_new) = 0.25$ and $P(gen\_basic\_rand) = 0.25$, since these values yielded the best results in earlier experiments [7].

2.4 Stopping Criterion

The stopping criterion for the algorithm presented in this paper is time. We believe that the questions we are trying to answer here should take into account testing time rather than the number of test cases because the testing process used throughout this paper is fully automatic. If manual pre- or post-processing per test case were required, the number of test cases would be much more significant. In the absence of such manual steps, the only limiting factor is time.

There is also a conceptual problem with using the number of executed test cases as stopping criterion. Most often the method under test calls other methods. Because every method execution contains its own oracle, not only top level method calls count as a test case. Of course some of the methods called might not be of interest for testing, but others will. It is hence not completely clear what the number of executed tests should refer to: the number of synthesized method invocations, the number of total method invocations, the number of method invocations on the system under test, or maybe some combination thereof.

3 Experimental Setup

3.1 AutoTest

AutoTest, the tool used to run the experiments presented in this paper, implements the input generation algorithm as described in §2.3. The tool is launched from the command line with a testing time and the names of the classes to test. It tests these classes for the given time by calling their methods with randomly generated objects as targets and parameters. It then delivers the results, which include minimized failure-reproducing examples, if any, and statistics about the testing session.

AutoTest is composed of two processes: the test generator, also called "driver," implements the testing strategy and issues simple commands (such as object creation and method invocation) to another process, an interpreter, which carries out the actual test execution. If failures occur during testing from which the interpreter cannot recover, the driver shuts it down and then restarts it, resuming testing where it was interrupted. This separation of the test execution from the test strategy logic improves the robustness of the process.

Restarting the interpreter has an important consequence: it triggers the reinitialization of the object pool. Subsequent method calls will not be able to use any of the objects created previously, but must start from an empty pool and build it anew.

When AutoTest tests a class, it tests all its methods. AutoTest keeps track of the number of times each method was called and has the following fairness policy: it tries to call each method once before it calls any of them a second time. To achieve this, it associates priorities with the methods and changes these priorities so that they reflect how often the method was called. An interpreter restart does not cause the resetting of these priorities, so the fairness criterion is preserved across multiple interpreter sessions.

Method *write_tests* depicted below contains the main loop of the testing strategy used in this paper. At each step it selects a method for testing and then causes the execution of this method.

```
write_tests (timeout):
  from
      initialize_pool
  until timeout
  loop
    m := choose (methods_under_test ())
    write_test_for_method (m)
  end
```

Method *initialize_pool* creates an empty pool of objects to be used for testing (both targets and parameters). The method *methods_under_test* returns the set of methods under test. The non-deterministic method *choose* selects an arbitrary element of a set or a list. The method *write_test_for_method* is described in figure 3.

Figure 2. Testing loop in a nutshell

3.2 Experiment

In the experiments, each of 27 classes was tested in 30 sessions of 90 minutes each, where in each session a different seed was used to initialize the pseudo-random number generator used for input creation. The total testing time was thus $30 * 27 * 90$ minutes $= 50.6$ days. All the tested classes were taken unmodified from the EiffelBase library version 5.6, which is used in almost all projects written in Eiffel,

The method *write_test_for_method* is responsible for generating a call to method *m*.
Notations: <*x1*, *x2*, *x3*> creates a list (ordered set) with the elements *x1*, *x2*, and *x3*. *list1* ... *list2* is the concatenation of *list1* and *list2* . Method *P*(*x*) non-deterministically evaluates to True with a probability of *x*. *type* (*m*) yields the type in which method *m* is contained.

```
write_test_for_method (m):
  ops := <>
  foreach ot from (<type (m)> ... param_types (m)) do
    if P (gen_new) then
        write_creation (ot)
    end
    ops := ops ... choose (conforming_objects (ot))
  end
   write_invoke_instruction (m, ops)
end
```

The method does two things for the target object and each method parameter:

1. With a probability of *P*(*gen_new*) it creates a new object and puts it into the pool.
2. It selects from the pool an arbitrary object conforming to the required type to be used as target object or parameter respectively. Note that, if an object is created in the previous step, this object is not necessarily selected in this step too. By chance an older object could be selected just as well.

Using the selected objects it serializes the actual invocation instruction as text to the output.
The method *write_creation* creates an object of type *t* and puts it into the pool:

```
write_creation (t):
  if is_basic_type (t) then
    if P (gen_basic_rand) then
      write_assignment (random_basic_object (t))
    else
      write_assignment (choose ( predefined_objects (t))
    end
  else
    c := choose (cons (t))
    ops := <>
    foreach ot from (param_types (c)) do
      if P(gen_new) then
          write_creation (ot)
      end
      ops := ops ... choose (conforming_objects (ot))
    end
     write_creation_instruction (c, ops)
  end
end
```

The method treats basic types (e.g. *INTEGER*, *DOUBLE*, *BOOLEAN*, ...) and reference types differently. For basic types, with a probability of *P*(*gen_basic_rand*), it creates an arbitrary random object of the required type using method *random_basic_object*. Since objects of all basic types are finite state, such a method is easy to devise using a random number generator. The alternative is to select an object from a predefined list that includes common values (e.g. 0, 1 for integers) and boundary values.
If the type to be created is not basic then the algorithm chooses randomly one of the available constructors for the required type. This constructor might also need arguments, so a similar algorithm to the one implemented in *write_test_for_method* is applied, with the difference that a constructor call needs only arguments and no target.

Figure 3. Methods *write_test_for_method* and *write_creation* which generate calls to invoke methods under test and create objects respectively.

Table 1. Metrics of the tested classes

	Average	Median	Minimum	Maximum
LoCs	477.67	366	62	2600
Methods	108.37	111	37	171
Attributes	6.26	6	1	16
Contracts	111.07	98	53	296
Faults	39.52	38	0	94

similar to the `system` library in Java or C#. The tested classes include widely used classes like *STRING* or *ARRAY* and also more seldom used classes such as *FIBONACCI* or *FORMAT_DOUBLE*. Table 1 shows various statistics of the code metrics of the classes under test: lines of code, number of methods, attributes, contract clauses and total number of faults found in the experiments. These statistics are meant to give an overview of the sizes of the classes. Detailed information about the class sizes is available online[2]. The number of methods, attributes and contracts includes the part that the class inherits from ancestors, if any.

During the testing sessions, AutoTest may trigger failures in the class under test and also in classes on which the tested class depends. There are two ways in which failures can be triggered in other classes than the one currently under test. First, a method of the class under test calls a method of another class, and the latter contains a fault which affects the caller. Second, the constructor of another class, of which an instance is needed as argument to a method under test, contains a fault.

AutoTest reports faults from the first category as faults in the class under test. This is because, although the method under test is not responsible for the failure, this method cannot function correctly due to a faulty supplier and any user of the class under test should be warned of this. Faults from the second category, however, are not counted. This is because in these experiments we focus on faults found in the class under test only. Such tests are nevertheless also likely to reveal faults (cf. the related analysis on the benefits of "interwoven" contracts [19]). How many of them are found there and how this impacts the predictability of random testing is a subject of further studies.

Computing infrastructure The experiments ran on 10 dedicated PCs equipped with Pentium 4 at 3.2GHz, 1Gb of RAM, running Linux Red Hat Enterprise 4 and ISE Eiffel 5.6. The AutoTest session was the only CPU intensive program running at any time.

[2] `http://se.inf.ethz.ch/people/ciupa/public/random_oo_testing_experiment.zip`

4 Discussion

4.1 Observations and Analysis

Over all 27 classes, the number of detected faults ranges from 0 to 94, with a median of 38 and a standard deviation of 28. In two of the classes (*CHARACTER_REF* and *STRING_SEARCHER*) the experiments did not uncover any faults. Figure 4 shows the median absolute number of faults detected over time for each class.

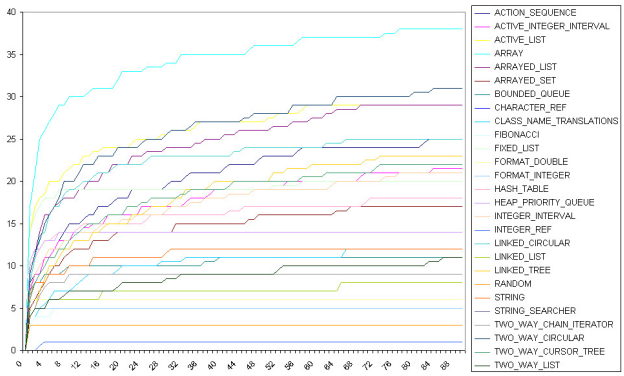

Figure 4. Medians of the absolute numbers of faults found in each class

In order to get aggregated results, we look at the *normalized* number of faults over time. For each class, we normalize by dividing the number of faults found by each test run by the total number of faults found for this particular class. The result is shown in Figure 5. When averaging over all 27 classes, 30% of the overall number of faults detected during our experiments are found after 10 minutes, as witnessed by the median of medians reaching .3 after 10 minutes in Figure 5. After 90 minutes, on average, an additional 8 percent of the overall number of randomly detected faults are found.

The main question is: **how predictable is random testing?** We consider two kinds of predictability: one that relates to the number of faults, and one that relates to the kind of faults and that essentially investigates if we are likely to detect the same faults, regardless of which of the thirty experiments is chosen. This provides insight into the influence of randomness (or, in more technical terms, the influence of the seed that initializes the pseudo-random number generator). Furthermore, we also consider how long it takes to detect a first fault and how predictable random testing is with respect to this duration.

In terms of *predictability of the number of detected distinct faults*, we provide an answer by considering the standard deviations of the normalized number of faults detected over time (Figure 6). With the exception of *INTEGER_REF*, an outlier that we do not show in the figure, the standard de-

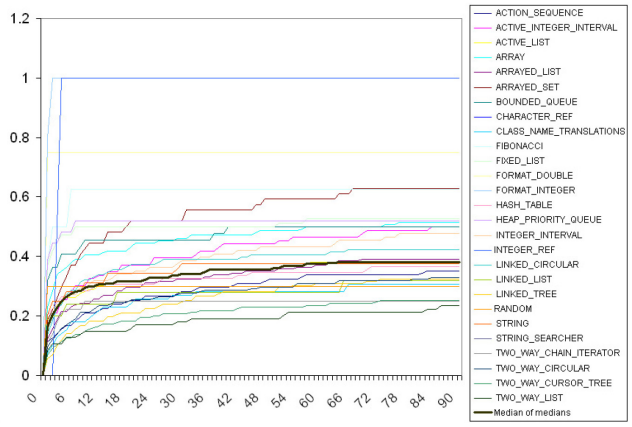

Figure 5. Medians of the normalized numbers of faults found in each class; their median

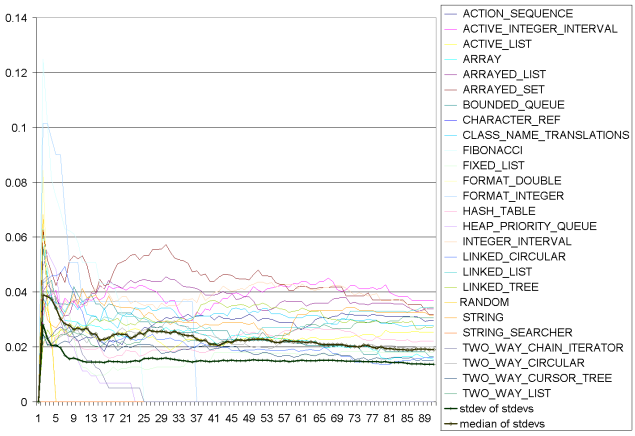

Figure 6. Standard deviations of the normalized numbers of faults found in each class; their median and standard deviation

viations lie roughly in-between 0.02 and 0.06, corresponding to 2% to 6% of the relative number of errors detected. Because we want to get one aggregated result across all classes, we display the median and the standard deviation of the standard deviations of the normalized number of detected faults in the same figure (median of standard deviations: upper thick line; standard deviation of standard deviations: lower thick line in Figure 6). The median of the standard deviations of the normalized numbers of detected faults decreases from 4% to 2% in the first 15 minutes and then remains constant. Similarly, the standard deviation of the standard deviations of the normalized number of detected faults linearly decreases from 3% to 1.5% after 10 minutes, and then remains approximately constant.

The median and standard deviation of the standard deviations being rather small suggests that random testing is, *in terms of the relative number of detected faults*, rather predictable in the first 15 minutes, and strongly predictable after 15 minutes. In sum, this somewhat counter-intuitively suggests that *in terms of the relative number of detected faults, random testing OO programs is indeed predictable*.

An identical relative number of faults does not necessarily indicate that the *same* faults are detected. If all runs detected approximately the same errors, then we could expect the normalized numbers of detected faults to be close to 1 after 90 minutes. This is not the case (median 38%) in our experiments: random testing exhibits a high variance in terms of the kind of detected failures, and thus appears *rather unpredictable in terms of the kind of the detected faults*.

Finally, when analyzing the results, we were surprised to see that *for 24 out of the 25 classes in which we found faults, at least one experiment detected a fault in the first second.* Taking a slightly different perspective, we could hence test any class thirty times, one second each. This means that within our experimental setup, random testing *is almost certain to detect a fault for any class within 30 seconds*. In itself, this is a rather strong predictability result.

This unexpected finding led us to investigate a question that we originally did not set out to answer: In terms of the efficiency of our technology, is there a difference between long and short tests? In other words, does it make a difference if we test one class once for ninety minutes or thirty times for three minutes?

To answer this question, we analyzed how the number of faults detected when testing for 30 minutes and changing the seed every minute compares to the number of faults found when testing for 90 minutes and changing the seed every 3 minutes and to the number of faults found when testing for 90 minutes without changing the seed, with longer test runs being an approximation of longer test sequences. Figure 7 shows the results of a class-by-class comparison.

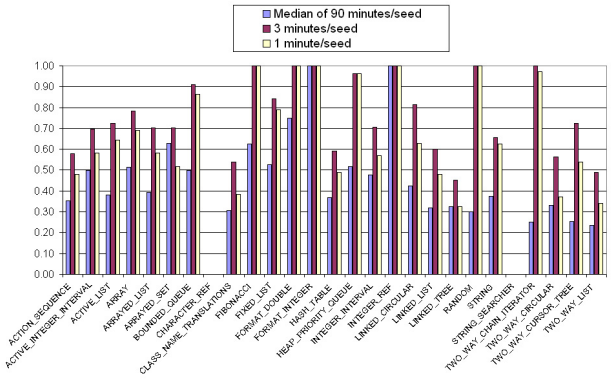

Figure 7. Cumulated normalized numbers of faults after 30*3 and 30*1 minutes; median normalized number of faults after 90 minutes

The results indicate that the strategy using each of the 30 seeds for 3 minutes (90 minutes altogether) detects more faults than using each of the thirty seeds for 1 minute (30 minutes altogether). Because the testing time is three times larger in the former when compared to the latter case, this is not surprising. Note, however, that the normalized number of faults is not three times higher. On more comparable grounds (90 minutes testing time each), *collating thirty times 3 minutes of test yields considerably better results than testing for 90 minutes.*

This suggests that short tests are more effective than longer tests. However, a more detailed analysis reveals that this conclusion is too simple. For the technical reasons described in §2, the interpreter needs to be restarted at least once during most of the experiments. In fact, there were only 60 experiments during which no interpreter restart occurred. Such a restart is similar to beginning a new experiment in that, as explained in §3.1, the object pool is emptied and must be constructed anew. Interpreter restarts do not, however, affect the scheduling of calls to methods under test: AutoTest preserves the fairness criteria and tries to call first methods that were tested the least up to that point. Because of these interpreter restarts, we cannot directly hypothesize on the length of test cases. In fact, we do not have any explanation of this stunning result yet, and its study is the subject of ongoing and future work.

4.2 Threats to the Validity of Generalizations of These Results

The classes used in the experiment belong to the most widely used Eiffel library and were not modified in any way. They are diverse both in terms of various code metrics and of intended semantics, but naturally their representativeness of OO software is limited.

A further threat to the validity of the present empirical evaluation is that the interpreter restarts trigger the emptying of the object pool. This puts a limit to the degree of complexity that the test inputs can reach. In our experiments, interpreter restarts occurred at intervals between less than a minute and over an hour. Even for the same class, these restarts occur at widely varying intervals, so that some sessions reach presumably rather complex object structures, and others only very simple ones.

AutoTest implements one of several possible algorithms for randomly generating inputs for OO programs. Although we tried to keep the algorithm as general as possible through various parameters, there exist other methods for generating objects randomly, as explained in §5. As such, the results of this study apply only to the specific algorithm together with specific choices for technical parameters (e.g., $P(gen\_new)$) for random testing implemented in AutoTest.

The full automation of the testing process – necessary also due to the sheer number of tests generated and executed in the experiment – required an automated oracle: contracts and exceptions. This means that naturally any fault which does not manifest itself through a contract violation or another exception could not be detected and included in the results presented here. Furthermore, we approximate faults by failures as described in §2.2. The automatic mapping from failures to faults could have led to faults being missed.

The results that we present relate to faults detected by random testing. Random testing may miss many more faults, and we do not know what the results would be if we compared the number of randomly detected faults to the number of faults that were detected manually, by random testing, and by other strategies.

The experiments reported here were performed only on classes "in isolation," not on a library as a whole or on an entire application. This means that the methods of these classes were the only ones tested directly. The methods that they transitively call are also implicitly tested. The results would probably be different for a wider testing scope, but timing constraints did not allow testing of entire applications and libraries.

As explained in §4.1, for 24 out of the 25 classes in which our experiments uncovered faults, there was at least one experiment in which the first fault for a class was found within the first second of testing. The vast majority of these faults are found in constructors when using either an extreme value for an integer or Void for a reference-type argument. It is thus questionable if these results generalize to classes which do not exhibit one of these types of faults. However, as stated above, in our experiment 24 out of the 27 tested classes did contain a maximum integer or Void-related fault.

5 Related Work

Much of the criticism of random testing in the literature stems from the intuition that, for most programs, this strategy stands little chance of coming across "interesting" and meaningful inputs. Several theoretical studies and reports on practical applications of the method contradict this intuition. Analytical studies by several authors [17, 22, 29] indicate that random testing can be as effective as (or even better than) partition testing. In contrast, Gutjahr shows that if the expected failure rates of the blocks of a partition are equal and one test is chosen per block, then the expected failure detection capability of partition testing will in general be better [15]. The assumption of equal expected failure rates is of course crucial here, and Gutjahr justifies it by the argument that if they were not equal, partition testing with one test per block would not be applied. However, *not knowing* failure rates does in general not imply that *their expectations are equal*. All these studies are theoretical and

focus on the comparison between partition and random testing, whereas the present study is purely empirical and aims at investigating several parameters about the performance of random testing alone.

On the practical side, random testing has been used to uncover bugs in Java libraries and programs [24], [9], in Haskell programs [8], in utilities for various operating systems [13], [21]. All these reports show that random testing does find defects in various types of software, but they do not investigate its predictability.

A comprehensive overview of random testing is provided by Hamlet [18]. He stresses the point that it is exactly the lack of system in choosing inputs that makes random testing the only strategy that can offer any statistical prediction of significance of the results. Hence, if such a measure of reliability is necessary, random testing is the only option. Furthermore, random testing is also the only option in cases when information is lacking to make systematic choices [16].

The interest in random testing of OO programs has increased greatly in recent years. Proof of this are the numerous testing tools using this strategy developed recently such as: JCrasher [9], Eclat [24], Jtest [1], Jartege [23], or RUTE-J [2]. The evaluations of these tools are focused on various quality estimation methods for the tools themselves: finding real errors in existing software (JCrasher, Eclat, RUTE-J), in code created by the authors (Jartege), in code written by students (JCrasher), the number of false positives reported (JCrasher), mutation testing (RUTE-J), code coverage (RUTE-J). As such, the studies of the behaviors of these tools stand witness for the ability of random testing to find defects in mature and widely used software and to detect up to 100% of generated mutants for a class. These studies do not, however, employ any statistical analysis which would allow drawing more general conclusions from them about the nature of random testing.

Several tools (RANDOOP [25], DART [14], ART [6], Agitator [3]) combine random testing with systematic approaches, in order to improve its efficiency by providing some guidance in the choice of inputs. The evaluations of RANDOOP, DART and ART use purely random testing as a basis for comparison: RANDOOP and DART are shown to uncover defects that random testing does not find, DART achieves higher code coverage than random testing, ART finds defects with up to 50% less tests than random testing. These results, although highly interesting in terms of comparing *different* testing strategies, do not provide much information about the performance of random testing itself or about the predictability of its performance.

A wide variety of other testing strategies also exists. Many strategies use symbolic execution to guide test case generation [30, 14, 10]. Tools like Korat [4] and Java PathFinder [28] offer bounded exhaustive exploration of test cases: Korat can generate all non-isomorphic inputs up to a given bound and JPF can explore bounded sequences of method calls exhaustively.

There are several studies which empirically compare the performance of various testing strategies against that of random testing. For example, Duran and Ntafos [12] compared random testing to partition testing and found that random testing can be as efficient as partition testing. Pretschner et al. [26] found that random tests perform worse than both model-based and manually derived tests. D'Amorim et al [11] compare the performance of two input generation strategies (random generation and symbolic execution) combined with two strategies for test result classification (the use of operational models and of uncaught exceptions). An interesting result of their study refers to the applicability of the two test generation methods that they compare: the authors could only run the symbolic execution-based tool on about 10% of the subjects used for the random strategy and, even for these 10% of subjects, the tool could only partly explore the code.

6 Conclusions and Future Work

The random generation of tests for Eiffel programs is particularly attractive because of the built-in oracle provided by the contracts. Random testing, by its very nature, is subject to random influences. Intuitively, choosing different seeds for different generation runs should lead to different results in terms of detected defects. Earlier studies had given initial evidence for this intuition. In this paper, we set out to do a systematic study on the predictability of random tests. To the best of our knowledge (and somewhat surprisingly), this question has not been studied before.

In sum, our main results are the following. Random testing is predictable in terms of the relative number of defects detected over time. In our experiments, random testing detects a defect within 30 seconds, (almost) regardless of the class under test, and regardless of the seed. On the other hand, random testing is much less predictable in terms of the kind of defects that are detected. We have commented on threats to validity of a generalization of these results.

Our study also led to one phenomenon that we cannot explain yet. If, for any class, we take the first three minute chunk of all thirty experiments and subsequently collate them, we obtain considerably better results than if we take the median of all 90-minute-runs for that class. This seems to suggest that short tests perform better than long tests, but because of frequent resets of the pool of objects used as inputs, this cannot be deduced.

Explaining this phenomenon is the focus of our current and future efforts. A systematic study on the effectiveness of short vs. long test runs is a further project and so is the repetition of our experiments with other classes and for

longer periods of time. Findings in this area could lead to conclusions as to whether random testing is "only" good at finding defects with a comparably simple structure, or if more complex defects can be detected as well. A related avenue of exciting research is concerned with the influence of a software system's coupling and cohesion on the efficiency of random tests.

References

[1] Jtest. parasoft corporation. http://www.parasoft.com/.

[2] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In *Proc. 1st Intl. Workshop on Random testing*, pages 36–45, 2006.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proc. Intl. Symp. on Software Testing and Analysis*, pages 123–133, 2002.

[5] P. Chalin. Are practitioners writing contracts? In *Springer LNCS 4157*, pages 100–113, 2006.

[6] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In M. J. Maher, editor, *Proc. Asian Computing Science Conference*, 2004.

[7] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proc. Intl. Symp. on Software Testing and Analysis*, pages 84–94, 2007.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[9] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[10] C. Csallner and Y. Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *Proc. Intl. Symp. on Software Testing and Analysis*, pages 245–254, July 2006.

[11] M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proc. Intl. Conf. on Automated Software Engineering*, pages 59–68, 2006.

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

[13] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows Systems Symposium*, Seattle, August 2000.

[14] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[15] W. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661674, September/October 1999.

[16] D. Hamlet. When only random testing will do. In *Proc. 1st Intl. Workshop on Random testing*, pages 1–9, 2006.

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

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

[19] Y. Le Traon, B. Baudry, and J. Jézéquel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006.

[20] A. Leitner and I. Ciupa. Autotest. `http://se.inf.ethz.ch/people/leitner/auto_test/`, 2005 - 2007.

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[22] S. Ntafos. On random and partition testing. In *Proc. Intl. Symp. on Software testing and analysis*, pages 42–48, 1998.

[23] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. Technical Report RR-1069-I, Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universite Joseph Fourier Grenoble I, June 2004.

[24] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. European Conf. on Object-Oriented Programming*, pages 504–527, 2005.

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. Intl. Conf. on Software Engineering*, pages 75–84, 2007.

[26] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One Evaluation of Model-Based Testing and its Automation. In *Proc. Intl. Conf. on Software Engineering*, pages 392–401, 2005.

[27] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato (New Zealand), April 2006.

[28] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proc. Intl. Symp. on Software testing and analysis*, pages 97–107, 2004.

[29] E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[30] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.