SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2011; **21**:3–28
Published online 6 July 2009 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.415

On the number and nature of faults found by random testing

I. Ciupa[1,*,†], A. Pretschner[2], M. Oriol[3], A. Leitner[4,‡] and B. Meyer[1]

[1] *Department of Computer Science, ETH Zürich, Switzerland*
[2] *Fraunhofer IESE and TU Kaiserslautern, Germany*
[3] *Department of Computer Science, University of York, U.K.*
[4] *Google Zurich, Switzerland*

SUMMARY

Intuition suggests that random testing should exhibit a considerable difference in the number of faults detected by two different runs of equal duration. As a consequence, random testing would be rather unpredictable. This article first evaluates the variance over time of the number of faults detected by randomly testing object-oriented software that is equipped with contracts. It presents the results of an empirical study based on 1215 h of randomly testing 27 Eiffel classes, each with 30 seeds of the random number generator. The analysis of over 6 million failures triggered during the experiments shows 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 experiment also suggests that the random testing quickly finds faults: the first failure is likely to be triggered within 30 s. The second part of this article evaluates the *nature* of the faults found by random testing. To this end, it first explains a fault classification scheme, which is also used to compare the faults found through random testing with those found through manual testing and with those found in field use of the software and recorded in user incident reports. The results of the comparisons show that each technique is good at uncovering different kinds of faults. None of the techniques subsumes any of the others; each brings distinct contributions. This supports a more general conclusion on comparisons between testing strategies: the *number* of detected faults is too coarse a criterion for such comparisons—the *nature* of faults must also be considered. Copyright © 2009 John Wiley & Sons, Ltd.

Received 15 August 2008; Revised 4 May 2009; Accepted 19 May 2009

KEY WORDS: random tests; object-oriented software; test selection strategies; empirical studies

1. INTRODUCTION

A substantial amount of research in software testing has been devoted to developing new or improving the existing fault detection strategies. It is often hard to know how these strategies and related tools perform in terms of their fault detection ability. The question has been addressed in two different ways. One is to assess the effectiveness and efficiency of a strategy in *absolute* terms.

*Correspondence to: I. Ciupa, ETH Zürich, Department of Computer Science, RZ J4, Clausiusstr. 59, 8092 Zurich, Switzerland.
†E-mail: ilinca.ciupa@inf.ethz.ch
‡Work carried out while this author worked at ETH Zurich.

Contract/grant sponsor: FhG Internal Programs; contract/grant number: Attract 692166

How many faults are detected? If a strategy is used till the end of generating tests [1], how much does it cost to generate and execute the tests? A second way is to perform *relative* assessments by comparing one strategy with others. Many researchers have followed the latter path, as witnessed by the large (and here necessarily incompletely cited) body of work [2–11]. Some studies have provided analytical answers such as subsumption relationships; others have focused on the number of faults experimentally detected by the different strategies. In sum, there is almost no conclusive evidence that one testing strategy would clearly outperform another one in terms of the number of detected faults.

Arguably, the least that one would expect from a testing strategy is that it performs better than random testing. Intuitively, random testing is the simplest form of generating tests and (seemingly) does not require too much intellectual or computational effort. Indeed, 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 [12]. 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 by reducing testing to robustness testing: random test case generation boils down to picking elements from the input domain and adding 'no exception' as expected output.

Testing object-oriented programs is slightly more challenging. This is 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 routines (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 routine is a non-trivial task.

This article analyzes one particular flavour of random testing for O-O software both in absolute and relative terms. In a first step, it analyzes several characteristics of random testing itself, namely its predictability in terms of both number and nature of detected faults, the average time it takes to detect a first failure, and the distribution of detected faults. These questions are motivated by the intuition that two distinct runs of a random test case generator—technically speaking, with two different seeds for the random number generator—should yield different results. It turns out that in the experiments, random testing is likely to detect the first failure within 30 s; that is it is highly predictable in terms of the relative number of detected faults; but that two different runs of a random testing tool are likely to reveal different faults. From an engineer's point of view, a high variance of random testing 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 with other strategies.

The finding that different runs of the random testing tool find approximately the same number yet different faults suggests that the number of faults is too coarse a criterion for the comparison of testing strategies. Consequently, a second part of the article compares the nature of faults revealed by random testing with the nature of faults revealed by manual testing and user incident reports. In these experiments, random testing turns out to be neither better nor worse than the other strategies. Each strategy finds different faults, which suggests that random testing should be used as a complement to these strategies, rather than as a competing technology.

More concretely, in terms of the *number of faults* detected, it first examines how similar the results of different test sessions of the same duration but with different seeds of the random number generator are. It then addresses the issue of predictability of random testing. Second, it sheds light on the *nature of faults* that random testing finds, in particular, when compared with manual testing and with user incident reports.

DOI: 10.1002/stvr

1.1. Experiments and results

The experiments presented here evaluate Eiffel programs. One distinctive feature of Eiffel programs is that they contain embedded executable specifications in the form of contracts[§]. Routine postconditions are one type of contracts. These naturally lend themselves to be used as oracles, with a level of abstraction somewhere in-between the concrete output and the abstract absence of exceptions [12]. Randomly generating test cases for Eiffel programs hence consists of (1) generating input objects for a routine to be tested and (2) adding the postcondition as the expected output.

The experiments use the AutoTest tool [14] to investigate the performance of random testing. AutoTest performs fully automated testing of contracted Eiffel programs: it calls the routines of the classes under test with randomly generated inputs (objects), and, if the preconditions of these routines are satisfied, it checks if contracts are fulfilled while running the tests. Any contract violation that occurs or any other thrown exception signals a fault.

AutoTest's strategy for creating inputs is not purely random: it is randomly combined with limit testing, as explained in Section 2. Previous experiments [15] have shown that this strategy is much more effective at uncovering faults than purely random testing at no extra cost in terms of execution time. It is thus more relevant to investigate the more effective strategy. As this strategy is still not only random-based but also uses special predefined values (which have a high impact on the results), in the rest of this article we use the term *random$^+$ testing* to refer to it.

1.1.1. Predictability of random$^+$ testing. The experiment for investigating the predictability of random$^+$ testing consisted of generating and running tests with AutoTest for 27 classes from a widely used Eiffel library, which was not modified in any way. Each class was tested for 90 min. To assess the predictability of the process, testing sessions ran for each class 30 times with different seeds for the pseudo-random number generator. The main results are the following:

1. On average 30% of the overall number of faults detected during the experiments are found after 10 min. After 90 min, on average, an additional 8% 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.

A package including the results and the source code of AutoTest is available online[¶]. It contains everything needed for the replication and extension of the experiments described in this article.

1.1.2. Nature of faults. The above results as well as some other earlier works [9, 15, 16] lead to the conjecture that the *number of faults* is too coarse a criterion for assessing and comparing testing strategies. Consequently, the present article investigates whether or not the *nature of faults* is a more suitable discriminator between different fault detection strategies.

This article evaluates three ways of revealing faults. These are manual unit testing, field use (with corresponding bug reports), and random$^+$ testing. They are representative of today's state of the art: the first two are widely used in the industry, and the last reflects the research community's current interest in automated testing solutions.

More specifically, AutoTest ran on 39 classes from EiffelBase, again without modification, and found a total of 165 faults. To investigate the performance of *manual testing*, these faults are compared with the faults found by the students who were explicitly asked to test three classes, two created by the authors and one slightly adapted from EiffelBase. *Faults in the field* are taken

[§] The widely spread view that developers do not see the advantages of contracts and will not go through the trouble of writing them is contradicted by a broad empirical study [13] that shows that programmers do write contracts, even if not complete ones.

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

DOI: 10.1002/stvr

from user-submitted bug reports on the EiffelBase library. These three ways of revealing faults are evaluated by comparing the number and distribution of faults they detect via a custom-made classification that contains 21 categories.

The results of this study provide more empirical evidence that different strategies do indeed uncover significantly different kinds of faults. Random$^+$ testing is particularly good at detecting problems found in specifications. It is not so good at detecting problems of overly strong preconditions, infinite loops, and 'semantic' problems as discussed below. It detects most of the faults uncovered by manual testing, plus some. This suggests that random$^+$ testing should be applied before human testers enter the loop. In addition, random$^+$ testing finds only a small percentage of user-reported faults; this suggests that random$^+$ testing cannot replace collecting bug reports from software users.

A more general conclusion is that testing strategies should indeed be compared in terms of the *nature* of faults and not only the *number* of faults that they find.

1.2. Contributions

Reflecting on the two sets of research questions, the contributions of the present work can be summarized as follows. First, this work presents empirical evidence on the predictability of random$^+$ testing. While the effectiveness of random testing has been studied before, there is no other investigation on its predictability. Such an investigation is important because being able to quantify the influence of randomness in the results of a testing session is necessary in order to make an informed decision about when to stop testing and to make estimations about the remaining faults that could be found by random testing. Naturally, predictability alone does not suffice for characterizing the performance of random testing or for comparing it with other testing strategies; having a predictable random testing process is, however, necessary for being able to compare the performance of random testing with that of another testing strategy and have significant results.

Second, fault classifications have previously been used to analyze the difference between inspections and software testing. Yet, this is the first study that (1) develops a classification of faults specifically targeted at contracted O-O software and (2) uses this classification to compare an automated random testing strategy with manual testing, and to furthermore compare testing results with faults detected in the field. The *nature* of faults has not been used when comparing random testing with other ways of detecting faults.

Combining these two parts of the analysis yields a comprehensive picture of the performance of random testing for object-oriented software and highlights its strengths and weaknesses.

Earlier descriptions of the experiments on predictability and the nature of faults have been provided in two conference publications [16, 17].

1.3. Overview

Section 2 provides the background for random$^+$ tests of O-O software and introduces the AutoTest tool. Section 3 describes the experiments and results that were used to assess the predictability of random$^+$ testing. Section 4 presents the experiments and results that relate to the nature of faults. In particular, it contains the classification of faults. After putting this work in context in Section 5, this article presents the associated conclusions in Section 6.

2. RANDOM TESTS FOR OBJECT-ORIENTED PROGRAMS

This section explains the testing technique used for the experiments described in this article. It first defines the notions of test case (Section 2.1) and fault and failure (Section 2.2) as used throughout this article. Section 2.3 explains the test case synthesis algorithm used. Finally, Section 2.4 presents AutoTest, the testing framework in which the algorithm is implemented.

2.1. Unit tests

All the tests performed in the experiments are unit tests of O-O software. It is hence important to start by defining and discussing the notions of unit, test input, and oracle in the context of O-O software.

1. In general, the purpose of *unit* testing is to verify the run of a certain program unit. In practice, the purpose of a unit test (xUnit-style) is most often to verify the run of a particular routine. Typically this is reflected in the names of manually written unit tests, when, for instance, a routine x is tested by the test routine named $test\_x$. What happens, however, if routine x itself calls other routines (e.g. routine y)? In a strict interpretation of unit testing, these other routines are not under test and must be replaced by simple stubs that are hard-coded to work with the given test only. In practice this does not always happen. The called routine (y) is stubbed only if it is non-trivial, if it conceptually belongs to another 'unit' or if it depends on an environment that is difficult to set up automatically. These criteria cannot be judged automatically and require the knowledge of the test engineer.
2. A unit test needs test *input*. Test input can be created either by directly writing bits to memory (similar to how previously saved objects are loaded into memory through deserialization) or by issuing a series of constructor calls that create and modify data through normal means. These two techniques have different advantages: creating the state directly gives complete control over the input creation, while creating the data through constructor calls yields data that is more likely to be relevant.
3. The *oracle* or the expected output part 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 fulfil for a particular input, it can specify conditions that the output should fulfil for any input, or it can be as abstract as specifying 'no exception' as the expected output. The more concrete the oracle, the easier it is to express complex scenarios. The more abstract the oracle, the easier it is to re-use for other tests and the less likely it is to introduce an implementation bias.

The experiments described in Section 3 of this article ran for 1215 h. For experiments of this size, it is important that *the notion of test case be chosen with automation and efficiency in mind*. Testing must be completely automated, requiring no human intervention and pre or postprocessing. For practical reasons, the following choices were made, with regard to the issues stated above:

Unit: In accordance with common practice, the routine is the unit under test. The test synthesizer creates test cases and the goal of each test case is to test one particular routine. No stubbing techniques were used because stubs cannot be automatically created.

Input: Input objects are created through regular constructor and routine 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 when writing arbitrary bits to memory. Hence creation through execution promises to be more efficient. In order to enforce more diversity, objects are also modified once they are constructed via regular routine calls. If objects were not modified, test cases would only be using objects in their initial state. In order to improve the efficiency of testing, objects are re-used from previous test cases for new tests to some degree. As tests are likely to modify these objects, this results in some diversification as well.

Oracle: In order to achieve full automation, contract-based oracles are used. A contracted program contains its specification in the form of routine pre and postconditions and class invariants. Preconditions are used as filters for invalid input: if a routine is called by the test and its precondition is violated, the test is invalid and hence worthless. Postconditions are used to detect failures: if the test calls a routine and satisfies its precondition, the routine body gets executed; if after the execution of the routine body the postcondition is violated, the test revealed a fault in the routine.

DOI: 10.1002/stvr

2.2. *Faults and failures*

In this article, the notions of faults and failures are largely determined by the contract-based oracle. In general, a *failure* is an observed difference between the actual and intended behaviours. An *error* is a program state that is not in accordance with the intended state. Errors may (but do not have to) lead to failures. In this article, every contract violation (except for the case when a test case directly violates the precondition of the routine under test) is interpreted as a failure. However, programmers are interested in *faults* in the software: wrong pieces of code that trigger the failures. Hence, an analysis of random testing should consider the detected faults, not the failures.

Mapping failures to faults is a part of the debugging process and is usually done by humans. This is the case for the results reported in Section 4, but for the experiment reported in Section 3, in which around 6 million failures were triggered, this was not possible. Instead results rely on an approximation that maps failures to faults 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 routine and the same class (contract violations result in exceptions as well). Throughout this article, the term 'fault' is used in this sense. Thus, the 6 million failures recorded in the experiment were mapped to a lot fewer faults in the code, because each fault triggered many failures at runtime.

Another note about identifying faults is necessary here. As explained above, no stubs are used for the tests, so a routine under test may not be able to function correctly due to another routine that it calls. Strictly speaking, the fault is not in the routine under test, but still this routine cannot function correctly (cannot fulfil its own contract) due to one of its suppliers. The routine under test is hence reported as being faulty (but not the called routine as well).

The contract-based oracle finds mismatches between the specification and the implementation. Deciding if a particular fault is due to incorrect code or an incorrect specification requires deciding whether there is a mismatch between the intended and the given specification. This is discussed in detail in Section 4.

2.3. *Test case synthesis algorithm*

As mentioned above, the units under test are routines of Eiffel classes. Assume that a routine m with return type R and formal arguments $p_1, \ldots, p_n$ of types $C_1, \ldots, C_n$ is defined in some class C. In order to test m, an object c of type C must be generated along with actual arguments $a_1, \ldots, a_n$ of types $C_1, \ldots, C_n$. Since a test case consists of both input and expected output, an object r of type R that represents the expected output is also generated. One can then execute $c.m(a_1, \ldots, a_n)$, compare the resulting object with r, and check if the effect of m on c and the a_i is as expected. The inputs (the target object c and the argument objects a_i) are generated through constructor calls, which themselves are routines and may need arguments; hence, the generation procedure is recursive.

As a program executes, objects are modified and may hence lead to interesting inputs for testing routines. In order not to use only freshly created objects when invoking routines, objects are stored in an object pool. Parameters for routine invocations are then created or chosen probabilistically: with a predefined probability, a new object is created; and with the complement of that probability, an existing object is picked from the pool. Over time, the object pool contains more and more complex objects and object structures that are used for testing. Further details about the input generation process are available in previous publications [16, 17].

As mentioned above the oracle is contract-based. Rather than generating an object r of type R and checking the effects of executing m on c and the a_i, the contracts are natural oracles: when the execution of $c.m(a_1, \ldots, a_n)$ finishes, AutoTest checks if the postcondition of m and the invariant of C are satisfied. In other words, the oracle is free, namely in the form of postconditions and invariants.

The generation of tests is guided by the selection of routines to be tested, and then by the creation or selection of argument objects. Because routines can be executed in arbitrary order and potentially infinitely, there is no natural notion of sampling tests (executions) of a program, and, consequently, this article cannot report on the statistical properties of this process.

2.4. AutoTest

The algorithm described above constitutes the backbone of a contract-based testing tool called AutoTest, which allows the easy plug-in of different testing strategies. The tool is launched from the command line with a testing time and the names of the classes to test. It then tests these classes according to the plugged strategy (*random$^+$ testing* in this article) for the given time. At the end of the testing session it delivers the results, which include minimized failure-reproducing examples, if any, and statistics about the testing session.

In order to increase robustness, 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 routine 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.

Restarting the interpreter has an important consequence: it triggers the reinitialization of the object pool (Section 2.3). Subsequent routine 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 routines. AutoTest keeps track of the number of times each routine was called and has the following fairness policy: it tries to call each routine once before it calls any of them a second time. To achieve this, it associates priorities with the routines and changes these priorities so that they reflect how often the routine was called. An interpreter restart does not cause the resetting of these priorities, so the fairness criterion is preserved across multiple interpreter sessions.

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 routine of the class under test calls a routine of another class, and the latter contains a fault that affects the caller. Second, the constructor of another class, of which an instance is needed as argument to a routine under test, contains a fault.

AutoTest reports faults from the first category as faults in the class under test. This is because, although the routine under test is not responsible for the failure, this routine 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 these experiments 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 [18]).

The stopping criterion used by AutoTest is time. The reason is that the questions to answer here should take into account testing time rather than the number of test cases because the testing process used throughout this article is fully automatic. If manual pre or postprocessing per test case was required, the number of test cases would be more significant. In the absence of such manual steps, the only limiting factor is time.

There is also a conceptual problem in using the number of executed test cases as the stopping criterion. Most often the routine under test calls other routines. Because every routine execution contains its own oracle, not only top level routine calls count as a test case. Of course some of the routines 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 routine invocations, the number of total routine invocations (i.e. including the routines that are called by the routine under test), the number of routine invocations on the system under test, or maybe some combination thereof.

3. NUMBER OF FAULTS

One of the research questions that this work aims to answer addresses the predictability of random$^+$ testing. Recall that testing a class means that all routines in this class are tested. If one particular run of the testing tool on a class revealed a number of faults in this class, is another run of the testing tool on that class, for the same duration but using a different seed for the pseudo-random

Table I. Metrics of the tested classes.

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

number generator, likely to reveal *a similar number of faults* and, furthermore, *the same faults* in the class under test? The experiment devised to answer these questions consisted in running AutoTest in several sessions on classes from the most widely used Eiffel library, without making any change to the classes, and comparing the results of 90 min long test sessions using different seeds for the pseudo-random number generator.

This section describes the experiments (Section 3.1), the results and their interpretation (Section 3.2), and finally discusses the threats to the validity of generalizations of the results (Section 3.3).

3.1. Experimental setup

In the experiments, each of the 27 classes was tested in 30 sessions of 90 min 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 $30*27*90\,\text{min}=50.6$ 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. This library is comparable with 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 I shows various statistics of the code metrics of the classes under test: lines of code, number of routines, attributes, number of lines of contract code (LOCC), and the 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[||]. The number of routines, attributes, and contracts includes the part that the class inherits from ancestors, if any.

The experiments were run on 10 dedicated PCs equipped with Pentium 4 at 3.2 GHz, 1 Gb 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.

3.2. Experimental results

Over all in 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 1 shows the median absolute number of faults detected over time for each class.

In order to get aggregated results, it is important to consider the *normalized* number of faults over time. This number is obtained for each class 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 2. When averaging over all 27 classes, 30% of the overall number of faults detected during the experiments are found after 10 min, as witnessed by the median of medians reaching 0.3 after 10 min in Figure 2. After 90 min, on average, an additional 8% points of the overall number of randomly detected faults are found.

The main question of this section is: *how predictable is random$^+$ testing?* Two kinds of predictability are considered: one that relates to the number of faults, and one that relates to the identity of faults and that essentially investigates if it is likely to detect the same faults, regardless

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

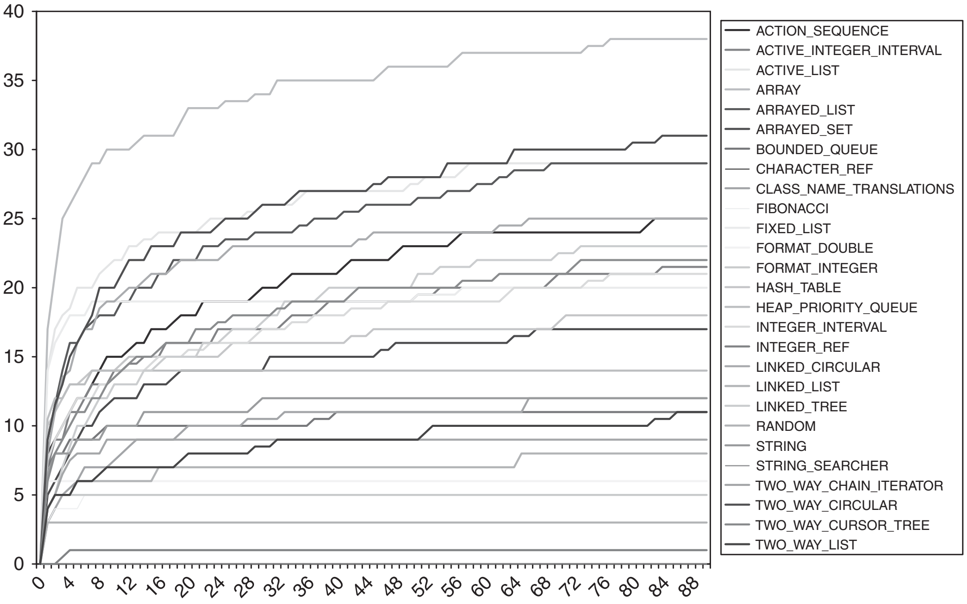

Figure 1. Medians of the absolute numbers of faults found in each class.

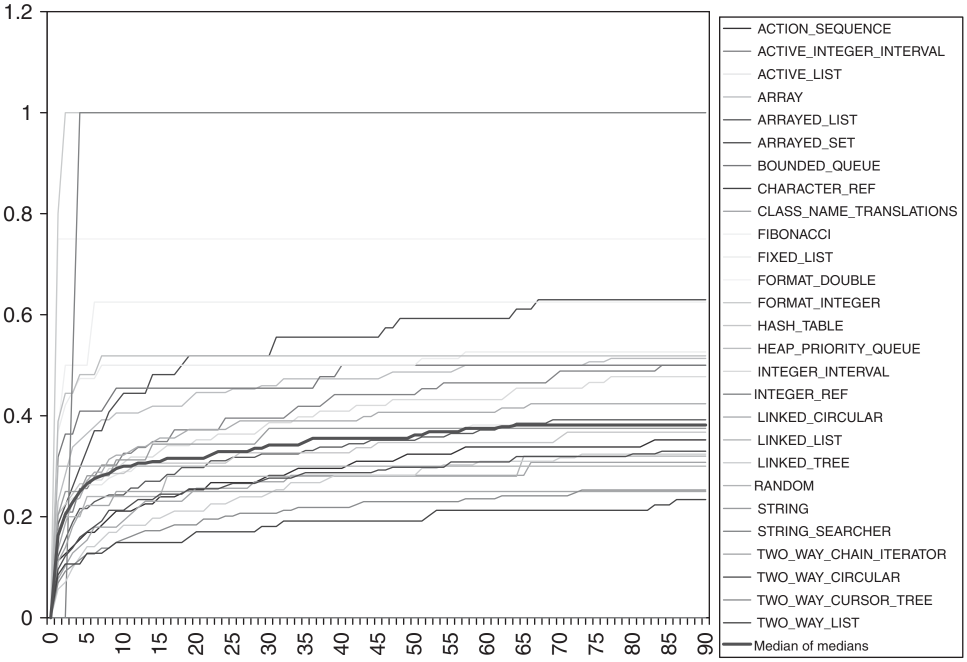

Figure 2. Medians of the normalized numbers of faults found in each class; their median.

of which of the 30 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, it also considers how long it takes to detect a first fault and how predictable random$^+$ testing is with respect to this duration.

To assess the *predictability of the number of detected distinct faults*, one can examine the standard deviations of the normalized number of faults detected over time (Figure 3). With the

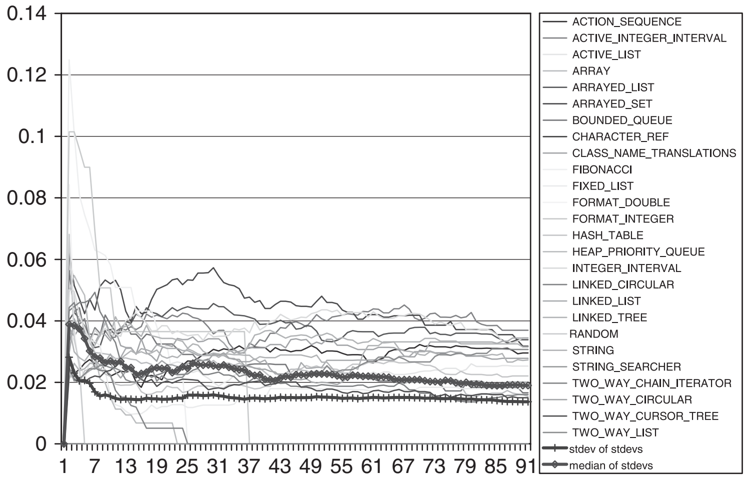

Figure 3. Standard deviations of the normalized numbers of faults found in each class; their median and standard deviation.

exception of *INTEGER_REF*, an outlier omitted in the figure, the standard deviations lie roughly in-between 0.02 and 0.06, corresponding to 2 to 6% of the relative number of errors detected. To present aggregated results across all classes, the median and the standard deviation of the standard deviations of the normalized number of detected faults are shown in the same figure (median of standard deviations: upper thick line, standard deviation of standard deviations: lower thick line in Figure 3). The median of the standard deviations of the normalized numbers of detected faults decreases from 4 to 2% in the first 15 min 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 min, and then remains approximately constant.

The median and standard deviation of the standard deviations being rather small suggests that different runs of the random testing tool uncover approximately the same numbers of faults; hence, random$^+$ testing is, *in terms of the relative number of detected faults*, rather predictable in the first 15 min, and strongly predictable after 15 min. In sum, this somewhat counter-intuitively suggests that *in terms of the relative number of detected faults, random$^+$ testing of O-O 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 the normalized numbers of detected faults would be close to 1 after 90 min. This is not the case (median 38%) in the experiments: random$^+$ testing exhibits a high variance in terms of the actual detected failures, and thus appears *rather unpredictable in terms of the actual detected faults*.

Thus, it seems that different runs of the random testing tool on the same class under test uncover approximately the same number of faults, but still partially different faults. A possible explanation for this result could take into account the ‘sizes’ of the uncovered faults: as defined by Offutt and Hayes [19], the *semantic size* of a fault in a program is ‘the relative size of the subdomain of *D* for which the output mapping is incorrect’, where *D* is the input domain of the program. Using this definition to explain the results of the present study, one could conjecture that different runs of the testing tool on the same class uncover faults with similar semantic sizes, and hence with approximately equal probabilities of being found through random testing. However, this semantic size of faults is only straightforward to calculate for programs with input domains having simple structures, as is the case, for instance, for primitive types. The inputs of object-oriented programs have inherently complex structures, which makes it very difficult, if at all possible, to calculate a size for the input domains and hence to calculate the semantic sizes of faults. It, therefore, seems dangerous to rely on such measures to explain the results of this study.

Examining the number of experiments (out of the total 30 runs per class) in which a particular fault was revealed sheds more light into the overlappings and differences, in terms of the faults

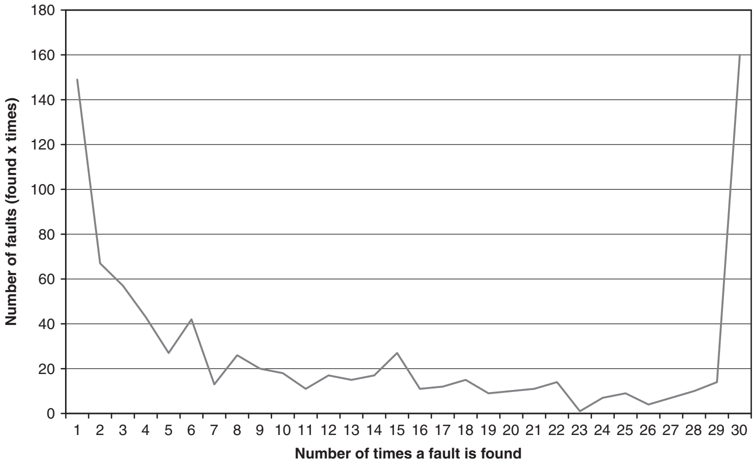

Figure 4. Number of faults found in 1, 2, ..., all 30 experiments for every class. Twenty-five per cent of all faults are uncovered by only 1 or 2 out of 30 experiments, 19% of all faults are uncovered in all 30 experiments.

found, between different runs of the test generator for each class. Each fault can be found 1 to 30 times; that is, it can be found only in 1 or in several and up to all 30 experiments (for a particular class). Figure 4 shows how many faults were found 1 to 30 times and it reveals an interesting tendency of random$^+$ testing: 25% of all faults are uncovered by only 1 or 2 out of 30 experiments, 19% of all faults are uncovered in all the 30 experiments.

The fact that many more faults are found in all 30 than in only 23 to 29 experiments may be surprising at first, but it is explainable through the nature of the testing process that was performed: random testing combined with special value testing. There are certain faults that AutoTest finds with high probability because these faults are evidenced through inputs (such as void pointers or minimum/maximum integer) that AutoTest uses with high probability. It is hence more likely that AutoTest finds, for example, a Void-related fault in all 30 experiments than that it finds it in only a subset of the 30 experiments.

Finally, when analyzing the results, it was surprising to see that *for 24 out of the 25 classes in which faults were found, at least one experiment detected a fault in the first second.* Taking a slightly different perspective, testing any class 30 times, 1 s each, could as well be performed. This means that within the experimental setup, random$^+$ testing *is almost certain to detect a fault for any class within 30 s*. In itself, this is a rather strong predictability result.

This unexpected finding raised another question: in terms of the efficiency of the technology, is there a difference between long and short tests? In other words, does it make a difference to test one class once for 90 min rather than 30 times for 3 min?

To answer this question, this article also analyzes how the number of faults detected when testing for 30 min and changing the seed every minute compares with the number of faults found when testing for 90 min and changing the seed every 3 min and with the number of faults found when testing for 90 min without changing the seed. Here, longer test runs serve as an approximation to longer test sequences. The comparison hence involves testing sessions of different durations: 30 × 1 vs 30 × 3 vs (median of) 1 × 90 min. The inclusion of the 30 × 1 min session allows investigating the effects of shorter test sessions—and produced unexpectedly good results.

Figure 5 shows the results of a class-by-class comparison. The numbers indicate that the strategy using each of the 30 seeds for 3 min (90 min altogether) detects more faults than using each of the 30 seeds for 1 min (30 min altogether). Because the testing time is three times larger in the former than in 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 min testing time each), *collating 30 times 3 min of test yields considerably better results than testing for 90 min.* Recall that the present approach to random testing relies on the notion of an object pool (Section 2.3). This object

DOI: 10.1002/stvr

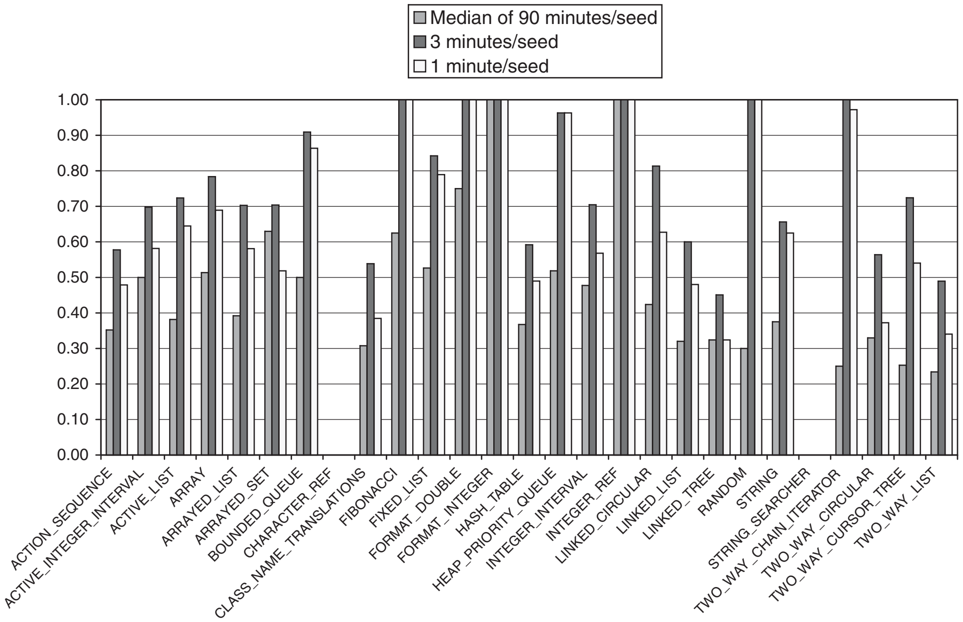

Figure 5. Cumulated normalized numbers of faults after $30*3$ and $30*1$ min; median normalized number of faults after 90 min.

pool constitutes (one part of) the state of the AutoTest tool, which explains why it is not necessarily the case that n sessions of duration m trigger the same distribution of failures as do m sessions of duration n.

This finding suggests that short tests are more effective than longer tests However, a more detailed analysis reveals that this conclusion is too simple. As it turns out, the interpreter needs to be restarted at least once during most of the experiments (cf. Section 2). 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 Section 2.4, the object pool is emptied and must be constructed anew. Interpreter restarts do not, however, affect the scheduling of calls to routines under test: AutoTest preserves the fairness criteria and tries to call first routines that were tested the least up to that point. Because of these interpreter restarts, it is difficult to directly hypothesize on the length of the test cases. In fact, there is no explanation so far of this stunning result, and its study is the subject of ongoing and future work.

3.3. Threats to validity

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 O-O software is limited.

A further threat to the validity of the present empirical evaluation is that the interpreter restarts to trigger the emptying of the object pool. This puts a limit to the degree of complexity that the test inputs can reach. In the experiments, the 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 O-O programs. Although the algorithm is as general as possible through various parameters, there exist other methods for generating objects randomly, as explained in Section 5. As such, the results of this study apply only to the specific algorithm together with specific choices for technical parameters for random^{+} testing implemented in AutoTest.

DOI: 10.1002/stvr

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 that does not manifest itself through a contract violation or another exception could not be detected and included in the results presented here. Furthermore, faults are approximated by failures as described in Section 2.2. The automatic mapping from failures to faults could have led to faults being missed.

The experiments reported here were performed only on classes 'in isolation,' not on an entire application. This means that the routines of these classes were the only ones tested directly. The routines 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.

The initial fault population naturally has an influence on the outcome of random tests and affects the results of the study. This experiment did not involve seeding any faults in the code on which AutoTest was run, but there is no information about how and to what extent the code was tested before; it is hence unknown how this previous testing affected the remaining fault population. In particular, it seems reasonable to assume that if the code had been random tested before, some of the faults found in the present experiment would have been removed and the results of this study would have been different.

As explained in Section 3.2, for 24 out of the 25 classes in which the 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 whether these results generalize to classes that do not exhibit one of these types of faults. However, as stated above, in the experiment 24 out of the 27 tested classes did contain a maximum integer or Void-related fault.

4. NATURE OF FAULTS

While the number of faults detected by random$^+$ testing is predictable, one must also examine if this strategy finds a superset or a subset of the faults found by human testers and users of the software.

This section compares, first, the faults found by regular users with faults found by random$^+$ testing. Second, it compares faults found by human testers with faults found by random$^+$ testing. While random$^+$ testing discovers many more faults than either of the other strategies, it does not discover the same kind of faults. In particular, there are categories of faults that are only found by random$^+$ testing and categories of faults that random$^+$ testing fails to uncover.

Several such categories are presented herein. Consequently, the section starts by presenting a classification of faults that was used to compare random$^+$ testing with the other two strategies (Section 4.1). It then describes the artifacts examined in the experiment and how the experiment was conducted (Section 4.2), and proceeds to the results of the experiment (Section 4.3) in terms of: (1) a comparison of the type of faults found by AutoTest and reported by users for the EiffelBase library; and (2) a comparison of the type of faults found by AutoTest and by manual testers for a set of classes created by the authors and one class taken almost verbatim from the EiffelBase library. The section then summarizes the results, discusses the most important findings, and concludes with a presentation of the threats to the validity of generalizing the results (Section 4.4).

4.1. Classifications of faults

In programming languages with support for embedding executable specifications, a fault can be characterized by two dimensions: *location* and *cause*. The *location* defines whether it occurs in the specification or in the implementation. The *cause* defines the real underlying issue. The following paragraphs discuss both the dimensions and introduce the resulting fault categories. The classification is not domain-specific. Although other fault classification schemes exist, as discussed in Section 5, there is no other scheme for contracted code.

DOI: 10.1002/stvr

4.1.1. Specification and implementation faults. In contract-equipped programs, the software specification is embedded in the software itself. Contract violations are one of the sources of failures. Hence, faults can be located both in the implementation and in the contracts.

A *specification fault* is a mismatch between the intended functionality of a software element and its explicit specification (in the context of this study, the contract). Specification faults reflect specifications that are not valid, in the sense that they do not conform to user requirements. The correction of specification faults requires changing the specification (plus possibly also the implementation) [9]. As an example, consider a routine `deposit` of a class `BANK_ACCOUNT` with an integer argument representing the amount of money to be deposited into the account. The intention is for that argument to be positive, and the routine only works correctly in that case. If the precondition of `deposit` does not list this property, the routine has a *specification fault*.

In contrast, an *implementation fault* occurs when a piece of software does not fulfil its explicit specification, here its contract. The correction of implementation faults never requires changing the specification. Suppose that the class `BANK_ACCOUNT` also contains a routine `add_value` that should add a value, positive or negative, to the balance of the account. If the precondition does not specify any constraint on the argument but the code assumes that it is a positive value, then there is a fault in the implementation.

The notion of specification fault assumes that it is possible to have access to the 'intended specification' or user requirements of the software: the real specification that it should fulfil. When analyzing the faults in real-world software, this is not always possible. Discussing it with the original developers is also in most cases difficult or even impossible. To infer the intended specification, one must rely on subjective evidence such as: the comments in the routines under test; the specifications and implementations of other routines in the same class; how the tested routines are called from various parts of the software system. This strategy resembles how a developer not familiar with the class would proceed in order to find out what the software is supposed to do.

4.1.2. Classification of faults by their cause. Some kinds of specification and implementation faults tend to recur over and over again in practice. Their study makes it possible to obtain a finer-grained classification by grouping these faults according to the corresponding human mistakes or omissions—their causes. By analyzing the cause of all faults encountered in the study, the categories described below emerged. The classification was created with practical applicability in mind and mainly focuses on either a mistake in the programmer's thinking or a misused programming mechanism.

4.1.2.1. Specification faults. An analysis of specification faults led to the following cause-based categories, grouped by the type of contract to which they apply.

1. Faults related to *preconditions*:
   - *Missing non-voidness precondition*: A precondition clause is missing, specifying that a routine argument, class attribute, or other reference denoted by an argument or attribute should not be void (null).
   - *Missing min/max-related precondition*: a precondition clause is missing, specifying that an integer argument, class attribute, or other integer denoted by an argument or attribute should have a certain value related to the minimum/maximum possible value for integers.
   - *Missing other precondition part*: a precondition is under-specified in another way than in the previous cases.
   - *Precondition disjunction due to inheritance*: with multiple inheritance it can be the case that a routine merges several others, inherited from different classes. In this case, the preconditions of the merged routines are composed using disjunction with the most current ones. Faults in this category appear because of this language mechanism (e.g. disjunction with *true* in case a precondition was not specified).
   - *Precondition too restrictive*: the precondition of a routine is stronger than it should be.

DOI: 10.1002/stvr

2. Faults related to the *postcondition* include *wrong postcondition* (the postcondition of a routine is incorrect) and *missing postcondition* (the postcondition of a routine was omitted).
3. Faults related to *class invariants* include only one kind: the *missing invariant clause*—a part of a class invariant is missing.
4. Faults related to `check`[**] assertions include only one kind: the *wrong check assertion*—the routine contains a `check` condition that does not necessarily hold.
5. Finally, the following faults apply to *all contracts*:
   - *Faulty specification supplier*: a routine used by the contract of the routine under test[††] contains a fault, which makes the contract of the routine under test incorrect.
   - *Calling a routine outside its precondition from a contract*: the fault appears because the contract of the routine under test calls another routine without fulfilling the latter's precondition.
   - *Min/max int-related fault in specification* (other than missing precondition): the specification of the routine under test lacks some condition(s) related to the minimum/maximum possible value for integers. (Examples so far do not cover floating-point computation.)

The categories in this classification have various degrees of granularity. The reason is that the classification was derived from faults obtained through widely different mechanisms: by AutoTest, by manual testers, by users of the software. The categories emerged by identifying recurring patterns in existing faults, rather than by trying to fit faults into a scheme defined *a priori*. Where such patterns could not be found, the categories are rather coarse-grained.

4.1.2.2. Implementation faults. The analysis of implementation faults led to the following cause-based categories:

- *Faulty implementation supplier*: a routine called from the body of the routine under test contains a fault, which does not allow the routine under test to function properly.
- *Wrong export status*: this category refers particularly to the case of creation procedures (constructors), which in Eiffel can also be exported as normal routines. The faults classified in this category are due to routines being exported as both creation procedures and normal routines, but which, when called as normal routines, do not fulfil their contract, as they were meant to be used only as creation procedures.
- *External fault*: Eiffel allows the embedding of routines written in C. This category refers to faults located in such routines.
- *Missing implementation*: the body of a routine is empty, often signaling an uncompleted attempt at top–down algorithm design.
- *Case not treated*: the implementation does not treat one of the cases that can appear, the result typically being that a necessary `if` branch is missing.
- *Catcall*: owing to the implementation of type covariance in Eiffel, the compiler cannot detect (in the Eiffel version used) some routine calls that are not available in the actual type of the target object. Such violations can only be detected at runtime. This class groups faults that stem from this deficiency of the type system.
- *Calling a routine outside its precondition from the implementation*: the fault appears because the routine under test calls another routine without fulfilling the latter's precondition.
- *Wrong operator semantics*: the implementation of an operator is faulty, in the sense that it causes the application of the operator to have a different effect than intended.
- *Infinite loop*: executing the routine can trigger an infinite loop, due to a fault in the loop exit condition.

[**] An Eiffel `check` instruction is similar to an 'assert' in C and C++. It states a condition that should be fulfilled at a certain point in the execution of a block of code. If contract monitoring is on and the condition does not hold, the execution triggers an exception.

[††] In Eiffel, contracts can include function calls.

DOI: 10.1002/stvr

Three of the above categories are specific to the Eiffel language and would not be directly applicable to languages that do not support multiple inheritance (precondition disjunction due to inheritance), covariant definitions (catcalls), or the inclusion of code written in other programming languages (external faults). All other categories are directly applicable to other object-oriented languages with support for embedded and executable specifications.

4.2. Experimental setup

Recall that the purpose of the experiment is to investigate how the nature of faults reported by random$^+$ testing differs from the nature of faults reported by regular users and the nature of faults found by manual testers. A significant difference in the distribution of faults in the categories presented above would prove that none of the techniques subsumes any other.

To see how random$^+$ testing performs, AutoTest was run on classes from the EiffelBase library (Section 3). Overall, the experiment randomly tested 39 classes from the 5.6 version of the library and found a total of 165 faults in them.

Bug reports from users of the EiffelBase library were then considered. From the database of bug reports, those referring to faults present in version 5.6 of the EiffelBase library were selected and those which were declared by the library developers to not be faults or those that referred to the .NET version of EiffelBase, which AutoTest cannot test, were excluded. The analysis hence refers to the remaining 28 bug reports fulfilling these criteria.

To determine how manual testing compares with random$^+$ testing, a competition for students of computer science at ETH Zurich was organized. Thirteen students participated in the competition. They were given three classes to test. The task was to find as many faults as possible in these three classes in 2 h. Two of the classes were written for the competition (with implementation, contracts, and purposely introduced faults from various of the above categories), and one was an adapted version of the STRING class from EiffelBase.

Table II shows some code metrics for these three classes: number of lines of code (LOC), number of lines of contract code (LOCC), and number of routines. A class that was significantly larger and more complex than the others was chosen intentionally, to see how the students would cope with it. Although such a class is harder to test, intuition suggests that it is more likely to contain faults. The students had varying experience in testing O-O software; most of them had at least a few lectures on the topic. Nine out of the thirteen students declared that they usually or always unit test their code as they write it. They were allowed to use any technique to find faults in the software under test, except for running AutoTest on it. Although they would have been allowed to use other tools (and this was announced before the competition), all the students performed only manual testing. In the end they had to produce test cases revealing the faults that they had found, through a contract violation or another exception.

4.3. Experimental results

4.3.1. Random$^+$ testing vs user reports. Table III shows the distribution of specification and implementation faults (1) found by random$^+$ testing (labelled 'AutoTest' in the table) 39 classes from the EiffelBase library and (2) recorded in bug reports from professional users. Note that the results in this table refer to more classes tested with AutoTest than for which there are user reports: even if there are no user reports on a specific class, the class may still have been used in the field.

Table II. Classes tested manually.

Class	LOC	LOCC	#Routines
MY_STRING	2444	221	116
UTILS	54	3	3
BANK_ACCOUNT	74	13	4

DOI: 10.1002/stvr

Table III. Random$^+$ testing vs user reports.

	Spec. faults	Implem. faults
AutoTest	103 (62.42%)	62 (37.58%)
User reports	10 (35.71%)	18 (64.29%)

Table IV. Random$^+$ testing vs user reports: specification and implementation faults.

		Number of faults		Percentage of faults	
Cause	Id	AutoTest	Users	AutoTest (%)	Users (%)
Specification faults					
Missing non-voidness precondition	S1	22	0	13.33	0.00
Missing min/max-related precondition	S2	23	0	13.94	0.00
Missing other precondition part	S3	28	3	16.97	10.71
Faulty specification supplier	S4	7	0	4.24	0.00
Calling a routine outside its precondition from a contract	S5	0	0	0.00	0.00
Min/max int related fault in spec (other than missing precondition)	S6	4	0	2.42	0.00
Precondition disjunction due to inheritance	S7	2	0	1.21	0.00
Missing invariant clause	S8	3	0	1.82	0.00
Precondition too restrictive	S9	0	2	0.00	7.14
Wrong postcondition	S10	12	2	7.27	7.14
Wrong check assertion	S11	2	0	1.21	0.00
Missing postcondition	S12	0	3	0.00	10.71
Specification faults total		103	10	62.42	35.71
Implementation faults					
Faulty implementation supplier	I1	47	0	28.48	0.00
Wrong export status	I2	0	2	0.00	7.14
External fault	I3	1	0	0.61	0.00
Missing implementation	I4	2	2	1.21	7.14
Case not treated	I5	4	7	2.42	25.00
Catcall	I6	3	1	1.82	3.57
Calling a routine outside its precondition from the implementation	I7	5	1	3.03	3.57
Wrong operator semantics	I8	0	1	0.00	3.57
Infinite loop	I9	0	4	0.00	14.29
Implementation faults total		62	18	37.58	64.29

Almost two thirds of the faults found by random$^+$ testing were located in the specification of the software, that is, in the contracts. This indicates that random$^+$ testing is especially good at finding faults in the contracts. In the case of faults collected from users' bug reports, the situation is reversed: almost two thirds of user reports refer to faults in the implementation.

Table IV presents a more detailed view of the specification and implementation faults found by AutoTest and recorded in users' bug reports, grouping the faults by their cause, as explained in Section 4.1.

This detailed comparison sheds more light on the differences between the faults reported by users and those found by automated testing, and exposes strengths and weaknesses of both the approaches. One difference that stands out involves faults related to extreme values (either Void references or numbers at the lower or higher end of their representation interval) in the specification. Around 30% of the faults uncovered by AutoTest are in one of these categories, whereas users do not report any such faults. Possible explanations are that such situations are not encountered in practice; that users do not consider them to be worth reporting; or that users rely on their intuition of the range of acceptable inputs for a routine, rather than the routine's precondition, and their intuition corresponds to the intended specification, not to the erroneous one provided in the contracts.

Symmetrically, AutoTest does not find some faults that users report. In a sense, this is not surprising as AutoTest does not take into account the operational profile of the tested classes and tests all input values equally, except for the limited set of special values that are tested with a higher priority. For example, a method using the speed of a car (in km/h) would not expect input values below -100 or above 2000 and most of the values would be comprised between 0 and 200 with a higher probability around 60. In such a case, AutoTest would test with priority inputs around 0 and extreme integer values. The rest of the input would be evenly spread out through the integer values and would thus be unlikely to belong to the interval between 10 and 200, which is the interval that users would test with priority.

A further difference results from AutoTest's ability to detect faults from the categories 'faulty specification supplier' and 'faulty implementation supplier.' They mean that AutoTest can report that certain routines do not work properly because they depend on other faulty routines. In the examined bug reports, users never recorded such faults: they only indicated the routine that contained the fault, without mentioning other routines that also did not work correctly because of the fault in their supplier. An important piece of information gets lost this way: after fixing the fault, there is no incentive to check whether the clients of the routine now work properly, meaning to check that the correction in the supplier allows the client to work properly too.

Random$^+$ testing is particularly bad at detecting some categories of faults: too strong preconditions, faults that are a result of wrong operator semantics, infinite loops, missing routine implementations. None of the 165 faults found by AutoTest and examined in this study belonged to any of the first three categories, but the users reported at least one fault in each. It is not surprising that AutoTest has trouble detecting such faults. First, if AutoTest tries to call a routine with a too strong precondition and does not fulfil it, the testing engine will simply classify the test case as invalid and try again to satisfy the routine's precondition by using other inputs. Second, AutoTest also cannot detect infinite loops: if the execution of a test case times out, it will classify the test case as 'bad response'; this means that it is not possible for the tool to decide if a fault was found or not—the user must inspect the test case and decide. Third, users of the EiffelBase library could report faults related to operators being implemented with the wrong semantics. Naturally, to decide this, it is necessary to know the intended specification of the operator. Finally, AutoTest also cannot detect that the implementation of a routine body is missing unless this triggers a contract violation. An automatic tool can of course, through code analysis, find empty routine bodies, but not decide if this is a fault. Note that in these cases, the overall number of detected faults is rather low, which suggests special care in generalizing these findings.

AutoTest was also run exclusively on the classes for which users reported faults to see if it would find those faults (except for three classes which AutoTest cannot currently process as they are either expanded or built-in). When run on each class in 10 sessions of 3 min, AutoTest found a total of 268 faults[‡‡]. Four of these were also reported by users, so 21 faults are solely reported by users and 264 solely by AutoTest. AutoTest detected only one of the 18 implementation faults (5%) reported by users and 3 out of the 7 specification faults (43%). While theoretically it could, AutoTest did not find the user-reported faults belonging to such categories as 'wrong export status' or 'case not treated'. Longer testing times might, however, have produced different results.

4.3.2. Random$^+$ testing vs manual testing. To investigate how AutoTest performs when compared with manual testers, AutoTest was run in 30 sessions of 3 min (where each session used a different seed for the pseudo-random number generator) on the three classes that were tested manually: a slightly modified version of the STRING class from EiffelBase (also tested in the experiment reported in Section 4.3.1) and two classes written by the authors. The reason for taking these

[‡‡] However, 183 of these faults were found through failures caused by the classes RAW_FILE, PLAIN_TEXT_FILE, and DIRECTORY through operating system signals and I/O exceptions, so it is debatable if these can indeed be considered as faults in the software.

two classes instead of classes from the EiffelBase library was the need to keep the code size relatively small and to reduce the classes' dependencies on the existing code; this somewhat not only simplified the testers' task, but also made it realistic to find faults in the given code in 2 h, the time limit given to the manual testers. These two classes could not be used for the experiment described in Section 4.3.1 because they had not been a part of an open source library or application for years and hence of course-lacked user bug reports.

Table V shows a summary of the results. It displays a categorization of the fault according to the classification scheme used in this article (the category ids are used here; they can be looked up in Table IV), the name of the class where a fault was found by either AutoTest or the manual testers, how many of the manual testers found the fault out of the total 13 and a percent representation of the same information, and finally, in the last column, x's mark the faults that AutoTest detected.

The table shows that AutoTest found 9 out of the 14 faults that humans detected and 2 faults that humans did not find. The two faults that only AutoTest found do not exhibit any special characteristics, but they occur in class MY_STRING, which is considerably larger than the other two classes. The conjecture is that, because of its size, students tested this class less thoroughly than the others. This highlights one of the clear strengths of the automatic testing tool: the sheer number of tests that it generates and runs per time unit and the resulting routine coverage.

Conversely, three of the faults that AutoTest does not detect were found by more than 60% of the testers. One of these faults is due to an infinite loop; AutoTest, as discussed above, classifies timeouts as test cases with a bad response and not as failures. The other two faults belong to the categories 'missing non-voidness precondition' and 'missing min/max-related precondition.' Although the strength of AutoTest lies partly in detecting exactly these kinds of faults, the tool fails to find them for these particular examples in the limited time it is given. This once again stresses the role that randomness plays in the approach, with both advantages and disadvantages.

4.3.3. Discussion. Three main observations emerge from the preceding analysis.

1. Random^{+} testing is good at detecting problems in specifications. It is particularly good with problems that are related to limit values. Problems of this kind are not reported in the field but tend to be caught by manual testers.
2. AutoTest is not good at detecting problems with too strong preconditions, infinite loops, missing implementations, and operator semantics. This is due to the very nature of automated random testing.
3. In a comparison between automated and manual testing (i.e., not taking into consideration bug reports), AutoTest detects almost all faults also detected by humans, plus a few others.

Table V. Random^{+} testing vs manual testing.

Id	Class	# testers	AutoTest
S1	BANK_ACCOUNT	8 (61.5%)	
S1	UTILS	5 (38.5%)	x
S1	MY_STRING	1 (7.7%)	x
S2	BANK_ACCOUNT	8 (61.5%)	
S2	UTILS	7 (53.8%)	x
S2	MY_STRING	1 (7.7%)	x
S2	MY_STRING	5 (38.5%)	x
S2	MY_STRING	1 (7.7%)	x
S2	MY_STRING	0 (0%)	x
S3	BANK_ACCOUNT	1 (7.7%)	x
S3	UTILS	4 (30.8%)	x
S10	MY_STRING	1 (7.7%)	
I2	BANK_ACCOUNT	4 (30.8%)	x
I6	MY_STRING	1 (7.7%)	
I7	MY_STRING	0 (0%)	x
I9	MY_STRING	9 (69.2%)	

> For model-based testing, this confirms the findings of an earlier study [9] and speaks strongly in favour of running the tool on the code before having it tested by humans. The human testers may find faults that the tool misses, but a great part of their work will be done at no other cost than CPU power.

AutoTest finds significantly more faults in contracts than in implementations. This might seem surprising, given that contracts are Boolean expressions and typically take up far fewer lines of code than the implementation (14% of the code on average in this study). Two questions naturally arise. First, are there more faults in contracts than in implementations, i.e. do the results obtained with AutoTest reflect the actual distribution of faults? Second, is it interesting at all to find faults in contracts, knowing that contract checking is usually disabled in production code?

There is no clear answer derived from the present set of results to the first question. There is no evidence that there are more problems in specifications than in implementations. The only thing to deduce is that random testing that takes special care of extreme values detects more faults in specifications than in implementations. Around 45% of the faults are uncovered in preconditions, showing that programmers often fail to specify correctly the range of inputs or conditions on the state of the input accepted by routines.

It is also important to point out that a significant proportion of specification errors are due to void-related issues, which are not present in newer versions of Eiffel, starting with 6.2 (Spring 2008). These implement the ‘attached type’ mechanism [20], which removes the problem by making non-voidness part of the type system and catches violations at compile time rather than runtime.

In terms of whether it is useful or interesting to detect and analyze faults in contracts, one must keep in mind that most often the same person writes both the contract and the body of a routine. A fault in the contract signals a mistake in this person’s thinking just as a fault in the routine body does. Once the routine has been implemented, client programmers who want to use its functionality from other classes look at its contract to understand under what conditions the routine can be called (expressed by its precondition) and what the routine does (the postcondition expresses the effect of calling the routine on the state). Hence, if the routine’s contract is incorrect, the routine will most likely be used incorrectly by its callers, which will produce a chain of faulty routines. The validity of the contract is thus as important as the correctness of the implementation.

The existence of contracts embedded in the software is a key assumption both for the proposed fault classification and for the automated testing strategy used. This is not too strong an assumption because it has been shown [13] that programmers willingly use a language’s integrated support for Design by Contract, if available.

The evaluation of the performance of random$^+$ testing performed here always considers the faults that AutoTest finds over several runs, using different seeds for the pseudo-random number generator. In the previous sections it was shown that random$^+$ testing is predictable in the relative *number* of faults it finds, but not in the *actual* faults it finds. Hence, in order to reliably assess the types of faults that random$^+$ testing finds, it is necessary to sum up the results of different runs of the tool.

In addition to pointing out strengths and weaknesses of a certain testing strategy, a classification of repeatedly occurring faults based on the cause of the fault also brings insights into those mechanisms of the programming language that are particularly error-prone. For instance, faults due to the wrong export status of creation procedures show that programmers do not master the property of the language that allows creation procedures to be exported both for object creation and for being used as normal routines.

Testing strategies can also be compared in terms of the perceived effect on the reliability of the faults they find. For countable and finite input domains, this effect can be measured automatically in terms of the semantic size of the faults, as proposed by Offutt and Hayes [19]. For input domains whose elements have arbitrary complexity, as is the case for object-oriented programs, this is not possible. An alternative in this case can be to ask software users to rate the effect on reliability of the various faults. This was unfortunately not possible in the case of the study presented here.

DOI: 10.1002/stvr

4.4. Threats to validity

The biggest threat to the generalization of the results presented here is the small size of the set of manually tested classes, the analyzed user bug reports, and the group of human testers participating in the study. Future work should expand this study to larger and more diverse code bases.

As explained in Section 4.2, the study only considered bug reports submitted by users for the EiffelBase library. Naturally, these are not all the faults found in field use, but only the ones that users took the time to report. Interestingly, for all but one of these reports the users set the priority to either 'medium' or 'high'; the severity, on the other hand, is 'non-critical' for seven of the reports and either 'serious' or 'critical' for the others. This suggests that even users who take the time to report faults only do so for faults that they consider important enough.

It was not possible to perform the study with professional testers, so bachelor and master students of computer science were used instead; to strengthen the incentive for finding as many faults as possible, this was a competition with attractive prizes for the top fault finders. In a questionnaire they filled in after the competition, four of the students declared themselves to be proficient programmers and nine estimated they had 'basic programming experience.' Seven of them stated that they had worked on software projects with more than 10 000 lines of code and the others had only worked on smaller projects. As mentioned in Section 4.2, two of the classes under test given to the students were written by one of the authors, who also introduced the faults in them. These faults were meant to be representative of the actual faults occurring in real software, so they were created as instances of various categories described in Section 4.1, but naturally the fact that they were seeded in code written by one of the authors introduces a bias. All these aspects limit the generality of the present conclusions.

A further threat to generalization is due to the peculiarities of the random testing tool used. AutoTest implements one particular algorithm for random testing and the results described here would probably not be identical for other approaches to the random testing of O-O programs (e.g. the one implemented in the JCrasher tool [21]). In particular, extreme values are used to initialize the object pool (Section 2). While void objects are rather likely to occur in practice, extreme integer values are not. In other words, the approach, as mentioned earlier, is not entirely random.

In addition as noted, compile-time removal of void-related errors will affect the results, for ISO Eiffel and other languages (such as Spec# [22]) that have the equivalent of an 'attached type' mechanism.

Another source of uncertainty is the assignment of defects to a classification. Finding a consistent assignment among several experts is difficult [23]. In this study, one person was assigned to this task. While this yields consistency, running the experiment with a different person might produce different results.

The programming language used in the study, Eiffel, also influenced the results. A few of the fault categories are closely related to the language mechanisms that are misused or that allow the fault to occur. This is to be expected in a classification of software faults based on the cause of the faults.

Finally, the 'size' of the faults in the various categories was not considered. As defined by Offutt and Hayes [19], the *semantic size* of a fault is the relative size of the input subdomain for which the fault triggers a failure at runtime. In other words, the size of a fault describes the effect of the fault on the failure rate of the application. It would surely be interesting to look into the failure rates for faults found through random testing, but a fundamentally different study than the one reported here would be necessary for investigating this question.

5. RELATED WORK

This section presents the research related to the contributions of this article. It starts with an overview of the state-of-the-art in random testing, then overviews some existing analytical and empirical comparisons of different testing strategies, and finally briefly presents other fault classification schemes proposed in the literature.

5.1. Random testing

In a comprehensive overview of random testing, Hamlet [24] 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 [25].

The intuition that, for most programs, random testing stands little chance of coming across 'interesting' and meaningful inputs is contradicted by several reports on practical applications of the method: random testing has been used to uncover faults in Java libraries and programs [21, 26], in Haskell programs [27], in utilities for various operating systems [28, 29]. All these reports show that random testing does find defects in various types of software, but they do not investigate its predictability.

Perhaps due to this, the interest in random testing of O-O programs has increased greatly in recent years. Proof of this are the numerous testing tools using this strategy developed recently such as: JCrasher [21], Eclat [26], Jtest [30], Jartege [31], or RUTE-J [32]. The evaluations of these tools are focused on various quality estimation methods for the tools themselves: finding real errors in the 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 behaviours 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 that would allow drawing more general conclusions from them about the nature of random testing.

Random input generation delivers the best results when combined with an automated oracle, due to the numerous and untargeted tests that it produces. For a human it would be tedious to wade through these tests, out of which only a small proportion are fault-revealing. The power of random testing can be fully exploited only if the pass/fail decision is automated. A great part of the existing work on fully automated testing [26, 31, 33–35] thus uses built-in test oracles in the form of contracts. These can either be written by developers or inferred by a tool [36] from runs of the system under test.

5.2. Comparisons of testing strategies

Several analytical studies [6, 8, 37] indicate that random testing can be as effective as (or even better than) partition testing; others make the point that, under rather specific conditions, partition-based testing will in general be at least as effective as random testing [5]. 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 the predictability of random testing and at comparing the faults that it finds to those found through manual testing and by users of the software.

Morasca and Serra-Capizzano [38] proposed a new method for comparing testing strategies in terms of the expected number of triggered failures, based only on knowledge of the total order or a hierarchical order of the failure rates of the subdomains of the input domain. This method allows comparing random testing, subdomain-based testing techniques, testing strategies mixing these two approaches, and adaptive testing techniques. While their work is purely analytical and applicable to a wide variety of testing methods (under the given conditions), the work presented here is empirical and makes no assumption about the failure rates of the software under test. It also addresses the question of the type of faults found by random testing, not only the number of such faults.

There are several empirical studies that compare the performance of various testing strategies against that of random testing. For example, Duran and Ntafos [7] compared random testing to partition testing and found that random testing can be as efficient as partition testing. Pretschner *et al.* [9] found that random tests perform worse than both model-based and manually derived tests. D'Amorim *et al.* [39] compare the performance of two input generation strategies (random generation and symbolic execution) combined with two strategies for test result classification (the

DOI: 10.1002/stvr

use of operational models and 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.

Numerous other studies [40–43] have compared structural and functional testing strategies as well as code reading. Most of them have used small programs with seeded faults and compared the results of two or three strategies. The four quoted studies compare the automated selection of test cases using the control flow or the all-uses—respectively mutation or the all-uses—criteria and their outcome in terms of faults uncovered by each strategy. None of these studies compares manual testing with automated techniques. The present study compares random$^+$ testing with manual testing and user bug reports; it seems that this is the first time that these three methods of identifying software faults are correlated.

5.3. Fault classification schemes

Many fault classification models have been proposed in the past [44–49]. Knuth [50] pioneered the work on classification of defects by defining nine categories reflecting the faults that occurred most often during the development of TeX.

The orthogonal defect classification (ODC) [44] combines two different classifications: defect types and defect triggers. In a sense the present classification is an ODC in itself but the present classification of defect types is finer while the defect location is simpler than defect triggers.

The IEEE classification [45] aims at building a framework for a complete characterization of the defect. It defines 79 hierarchically organized *types* that are sufficient to understand any defect. In the present case, using such categories would not have helped significantly because these categories do not reflect the particular constructs of contract-enabled languages.

Ma *et al.* [51] define fault categories for Java programs and relate them to mutation operators. It is unclear how the mapping from failures to these fault categories could be done automatically, and the categories do not take contracts into account.

The classification probably most similar to the one presented here is that by Basili *et al.* [41–43], organized in two dimensions: whether the fault is an omission or a commission fault, and to which of six possible types it belongs. The present classification takes into account specifications (contracts) and is more fine-grained.

Bug patterns [52, 53] are also related to this work. The present approach has to cope with different constructs and thus defines categories adapted to Eiffel programs, taking into account contracts and multiple inheritance.

6. CONCLUSIONS

In spite of several decades of research on the subject matter, there still is no conclusive evidence on when to use a given test selection or assessment criterion, or when one criterion would outperform another criterion. It is a long-term research goal to provide such evidence.

This problem can be tackled from different angles. One is to analyze selection or assessment criteria in isolation. Measures of interest include the cost and the predictability with respect to the number and kind of revealed faults. Because test selection criteria define an abstraction on all possible executions of a system, many distinct test suites can satisfy one criterion. Whether or not all these test suites detect the same faults is not always clear, which, among other things, constitutes a problem when it comes to test suite minimization. In other words, predictability is not only an issue with random testing, but likely also with, say, coverage criteria that relate to the control flow. This kind of property is relevant when it comes to deciding which strategy should be applied in a given development context.

From another angle one can compare *different* selection criteria. Information on isolated criteria is useful; information that allows one to compare two selection criteria is even more useful. The results presented in this article—and by other researchers as well—seem to indicate that the number

DOI: 10.1002/stvr

of faults detected by one strategy is too weak a criterion. The conjecture is that this may be one reason why there are no compelling studies that would show the superiority of one criterion over another. Instead, the kind of faults should matter as well.

More concretely, the work presented in this article is on random$^+$ testing, a combination of random with limit testing (and it is not completely evident whether the results generalize to vanilla random testing). Random$^+$ tests are appealing because they are comparably easy and cheap to generate. They are particularly attractive for Eiffel programs because of the built-in oracle that is 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 the detected defects. Earlier studies had given initial evidence for this intuition. In the first part of the work presented in this article, a systematic study on the predictability of random tests is presented. Somewhat surprisingly it seems that this question has not been studied before.

In sum, the main predictability results are the following. Random testing is predictable in terms of the relative number of defects detected over time. Yet, different runs of the tool reveal different faults. The present article explained the threats to validity of a generalization of these results. An analytical approach to deriving these results may also be possible and will be investigated in future work.

The second question concerns the nature of faults that random$^+$ testing finds. Moreover, it is interesting to consider whether and how these differ from faults found by human testers and by users of the software. The experiments presented in this article then suggest that the three mentioned strategies for finding software faults have different strengths and applicability. None of them subsumes any other in terms of performance. Random$^+$ testing with AutoTest has the advantage of being completely automatic. Humans, however, find faults that AutoTest misses. This is shown both by the examined user bug reports and by asking testers to test some code on which AutoTest was run, and by subsequently comparing the results. This last experiment also proved that AutoTest finds faults that testers miss. The conclusion is that random$^+$ tests should be used alongside with manual tests. Given the earlier results on comparing different QA strategies, this is not surprising, but it seems that there are no other systematic studies that showed this for random testing. Threats to generalizing these results are discussed in Section 4.4.

The comparative analysis is based on a classification of faults that is not specific to one particular application domain. For specification faults, it covers typical problems with pre and postconditions and invariants that are too weak, for example by not taking extreme values into account, or too strong. For implementation faults, it covers a few general problems such as missing cases or infinite loops, and others that relate to the specifics of Eiffel. It is not possible to assert that the classification is complete or the only possible one; it is the result of analyzing the faults that were encountered. It is likely that this classification, or some variant of it, can be used for future experiments on comparing strategies for finding faults.

The results of research into randomly testing Eiffel programs can also be used for investigating the benefits of using contracts [18] and how to improve contracts, possibly based on specification patterns. Future work in this direction will require performing more experiments of the kind presented in this article, adjusting the classification, and comparing concrete testing strategies, such as partition-based testing, or usage-profile-based testing, rather than the admittedly underspecified 'manual testing strategy.'

The experiments also revealed one phenomenon that cannot be explained yet. If, for any class, the first 3 min chunk of all 30 experiments are taken and subsequently collated, results are considerably better than when taking the median of all 90 min 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 future efforts. A systematic study on the effectiveness of short vs long test runs is a further project and so is the repetition of the experiments with other classes and for longer periods of time. The definition of metrics for test cases that not only take into account the length of a test case, but also the complexity of the generated objects is also to be considered. 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 effectiveness and efficiency of random tests.

ACKNOWLEDGEMENTS

This work was supported by the FhG Internal Programs under Grant No. Attract 692166.

REFERENCES

1. Zhu H, Hall P, May J. Software unit test coverage and adequacy. *ACM Computing Surveys* 1997; **29**(4):366–427.
2. Frankl P, Weyuker E. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering* 1998; **14**(10):1483–1498.
3. Frankl P, Weiss S. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering* 1993; **19**(8):774–787.
4. Hutchins M, Foster H, Goradia T, Ostrand T. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. *Proceedings of ICSE*, Sorrento, Italy, 1994; 191–200.
5. Gutjahr W. Partition testing versus random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering* 1999; **25**(5):661–674.
6. Weyuker E, Jeng B. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering* 1991; **17**(7): 703–711.
7. Duran J, Ntafos S. An evaluation of random testing. *IEEE Transactions on Software Engineering* 1984; **SE-10**(4):438–444.
8. Hamlet D, Taylor R. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering* 1990; **16**(12):1402–1411.
9. Pretschner A, Prenninger W, Wagner S, Kühnel C, Baumgartner M, Sostawa B, Zölch R, Stauner T. One evaluation of model-based testing and its automation. *Proceedings of ICSE*, St. Louis, MO, U.S.A., 2005; 392–401.
10. Heimdahl M, George D, Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? *Proceedings of the Eighth IEEE High Assurance in Systems Engineering Workshop*, Tampa, FL, February 2004.
11. Bernard E, Legeard B, Luck X, Peureux F. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software—Practice and Experience* 2004; **34**(10):915–948.
12. Utting M, Pretschner A, Legeard B. A taxonomy of model-based testing. *Technical Report 04/2006*, Department of Computer Science, The University of Waikato, New Zealand, April 2006.
13. Chalin P. *Are Practitioners Writing Contracts* (*Lecture Notes in Computer Science*, vol. 4157)? Springer: Berlin, 2006; 100–113.
14. Leitner A, Ciupa I. AutoTest. 2005–2009. Available at: http://se.ethz.ch/research/autotest/ [11 June 2009].
15. Ciupa I, Leitner A, Oriol M, Meyer B. Experimental assessment of random testing for object-oriented software. *Proceedings of ISSTA*, London, U.K., 2007; 84–94.
16. Ciupa I, Pretschner A, Leitner A, Oriol M, Meyer B. On the predictability of random tests for object-oriented software. *Proceedings of the First International Conference on Software Testing, Verification and Validation* (*ICST'08*), Lillehammer, Norway, 2008; 72–81.
17. Ciupa I, Meyer B, Oriol M, Pretschner A. Finding faults: Manual testing vs. random testing+ vs. user reports. *Proceedings of the 19th International Symposium on Software Reliability Engineering*, Redmond, Seattle, U.S.A., 2008; 157–166.
18. Le Traon Y, Baudry B, Jézéquel J. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering* 2006; **32**(8):571–586.
19. Offutt AJ, Hayes JH. A semantic model of program faults. *ISSTA '96*: *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM Press: New York, NY, U.S.A., 1996; 195–200.
20. Meyer B. Attached types and their application to three open problems of object-oriented programming. *Proceedings of ECOOP* (*Lecture Notes in Computer Science*, vol. 3586). Springer: Berlin, 2005; 1–32.
21. Csallner C, Smaragdakis Y. JCrasher: An automatic robustness tester for Java. *Software*: *Practice and Experience* 2004; **34**(11):1025–1050.
22. Barnett M, Leino KRM, Schulte W. The spec# programming system: An overview. *Proceedings of CASSIS* (*Lecture Notes in Computer Science*, vol. 3362). Springer: Berlin, 2004.
23. Henningsson K, Wohlin C. Assuring fault classification agreement—An empirical evaluation. *Proceedings of the International Symposium on Empirical Software Engineering*, Redondo Beach, CA, U.S.A., 2004; 95–104.
24. Hamlet R. Random testing. *Encyclopedia of Software Engineering*, Marciniak J (ed.). Wiley: New York, 1994; 970–978.

DOI: 10.1002/stvr

25. Hamlet D. When only random testing will do. *Proceedings of the First International Workshop on Random Testing*, Portland, Maine, U.S.A., 2006; 1–9.
26. Pacheco C, Ernst MD. Eclat: Automatic generation and classification of test inputs. *Proceedings of ECOOP*, Glasgow, Scotland, U.K., 2005; 504–527.
27. Claessen K, Hughes J. QuickCheck: A lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices* 2000; **35**(9):268–279.
28. Forrester JE, Miller BP. An empirical study of the robustness of Windows NT applications using random testing. *Fourth USENIX Windows Systems Symposium*, Seattle, August 2000.
29. Miller BP, Fredriksen L, So B. An empirical study of the reliability of unix utilities. *Communications of the ACM* 1990; **33**(12):32–44.
30. Jtest. Parasoft Corporation. Available at: http://www.parasoft.com/ [11 June 2009].
31. Oriat C. Jartege: A tool for random generation of unit tests for java classes. *Technical Report RR-1069-I*, CNRS, Universite Joseph Fourier Grenoble I, June 2004.
32. Andrews JH, Haldar S, Lei Y, Li FCH. Tool support for randomized unit testing. *Proceedings of the First International Workshop on Random Testing*, Portland, Maine, U.S.A., 2006; 36–45.
33. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, Italy, 2002; 123–133.
34. Csallner C, Smaragdakis Y. DSD-crasher: A hybrid analysis tool for bug finding. *Proceedings of the International Symposium on Software Testing and Analysis*, Portland, Maine, U.S.A., July 2006; 245–254.
35. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Proceedings of the ICSE*, Minneapolis, MN, U.S.A., 2007; 75–84.
36. Ernst MD. Dynamically discovering likely program invariants. *PhD Thesis*, Department of Computer Science and Engineering, University of Washington, August 2000.
37. Ntafos S. On random and partition testing. *Proceedings of the International Symposium on Software Testing and Analysis*, Clearwater Beach, FL, U.S.A., 1998; 42–48.
38. Morasca S, Serra-Capizzano S. On the analytical comparison of testing techniques. *ISSTA '04*: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM: New York, NY, U.S.A., 2004; 154–164.
39. d'Amorim M, Pacheco C, Marinov D, Xie T, Ernst MD. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. *Proceedings of the ASE*, Tokyo, Japan, 2006; 59–68.
40. Frankl PG, Weiss SN, Hu C. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software* 1996; **38**:235–253.
41. Basili V, Selby R. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering* 1987; **13**(12):1278–1296.
42. Kamsties E, Lott C. An empirical evaluation of three defect-detection techniques. *Proceedings of the ESEC*, Sitges, Spain, 1995; 362–383.
43. Wood M, Roper M, Brooks A, Miller J. Comparing and combining software defect detection techniques: A replicated empirical study. *SIGSOFT Software Engineering Notes* 1997; **22**(6):262–277.
44. Chillarege R, Bhandari I, Chaar J, Halliday M, Moebus D, Ray B, Wong M-Y. Orthogonal defect classification—A concept for in-process measurements. *IEEE Transactions on Software Engineering* 1992; **18**(11):943–956.
45. IEEE standard classification for software anomalies. *IEEE Std 1044-1993*, 2 June 1994.
46. Lutz R. Targeting safety-related errors during software requirements analysis. *Proceedings of the ACM SIGSOFT FSE*, Cambridge, U.K., 1993; 99–106.
47. Gray J. Why do computers stop and what can be done about it? *Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, U.S.A., 1986; 3–12.
48. Beizer B. *Software Testing Techniques*. Wiley: New York, NY, U.S.A., 1990.
49. Ploski J, Rohr M, Schwenkenberg P, Hasselbring W. Research issues in software fault categorization. *SIGSOFT Software Engineering Notes* 2007; **32**(6):6.
50. Knuth DE. The errors of tex. *Software Practice and Experience* 1989; **19**(7):607–685.
51. Ma Y, Kwon Y, Offutt J. Inter-class mutation operators for java. *Proceedings of the ISSRE*, Annapolis, MD, U.S.A., 2002; 352–366.
52. Allen E. *Bug Patterns in Java*. APress L. P.: Berkeley, CA, U.S.A., 2002.
53. Hovemeyer D, Pugh W. Finding bugs is easy. *SIGPLAN Notices* 2004; **39**(12):92–106.

DOI: 10.1002/stvr