Finding Faults: Manual Testing vs. Random Testing+ vs. User Reports

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

Abstract

The usual way to compare testing strategies, whether theoretically or empirically, is to compare the number of faults they detect. To ascertain definitely that a testing strategy is better than another, this is a rather coarse criterion: shouldn't the nature of faults matter as well as their number? The empirical study reported here confirms this conjecture. An analysis of faults detected in Eiffel libraries through three different techniques—random tests, manual tests, and user incident reports—shows that each is good at uncovering significantly different kinds of faults. None of the techniques subsumes any of the others, but each brings distinct contributions.

1 Introduction

A substantial amount of research in software testing has been devoted to developing new or improving existing fault detection strategies. It is unclear, however, how related tools and strategies compare in terms of their fault detection ability. Many studies [19, 20, 17, 27, 22, 38, 15, 23, 36, 24, 7] have tried to answer this question. While some studies have provided analytical answers such as subsumption relationships, others have focused on the number of faults detected by the different strategies. In sum, none of the latter studies conclusively shows that one testing strategy clearly outperforms another in terms of the number of detected faults, not even in one restricted application domain.

Earlier work [11, 12, 36] has led us to conjecture that the *number of faults* is too coarse a criterion for assessing testing strategies. Consequently, in the experiments described in this paper, we investigate whether or not the *kind of faults* is a more suitable discriminator between different fault detection strategies. This work's main result is empirical evidence that different strategies do indeed uncover significantly different kinds of faults. This complements the seminal work by Basili et al. [5] who compared the types of faults found by code reading, manual functional testing, and manual structural testing. It also fits smoothly with our previous work on model-based testing that exhibited large differences between faults found with random testing and with other strategies [36] (where we acknowledged but did not analyze the differences).

The three fault detection strategies analyzed in this paper are manual unit testing, field use (with corresponding bug reports), and automated random testing. They are representative of today's state of the art: the first two are widely used in industry, and the last one reflects one of the research community's current focuses on automated testing solutions. Although random testing is only one of the trends in current research, it is attractive because of its simplicity and because generating test cases is comparatively cheap. Moreover, there is no conclusive evidence that random testing is any worse at finding faults than other automated strategies.

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 [8, 14, 33, 34, 32] thus uses built-in test oracles in the form of contracts. These can either be written by developers—programmers actually do write contracts when possible [9]—or inferred by a tool [16] from runs of the system under test. Preconditions, postconditions and class invariants provide an automated oracle with a level of abstraction [37] that is usually far more fine-grained than the simple robustness assessment that can be obtained from checking whether or not the system crashed. Satisfying preconditions is a theoretical challenge [3], but not necessarily a practical one.

In the experiment presented here, we used the AutoTest tool [29] for investigating the performance of random testing. AutoTest performs fully automated testing of contracted O-O software: it calls the routines (methods) 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 random combined with limit testing, as explained in §2. Previous experiments [11] 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, so we consider it more relevant to investigate the more effective strategy. As this strategy is still random-based but also uses special predefined values (which have a high impact on the results), in the rest of this paper we will refer to it as *random testing+*. We ran AutoTest on 39 classes from a widely-used Eiffel library, called EiffelBase, which we did not modify in any way. AutoTest found a total of 165 faults in these classes. To investigate the performance of *manual testing*, we analyzed the faults found by students who were explicitly asked to test three classes, two created by us and one slightly adapted from EiffelBase. *Faults in the field* are taken from user-submitted bug reports on the EiffelBase library. We evaluated these three ways of detecting faults by comparing the number and distribution of faults they detect via a custom-made classification that contains 21 categories.

Contributions. Fault classifications have previously been used to analyze the difference between inspections and software testing. Yet, as far as we know, this is the first study that (1) develops a classification of faults specifically adapted to contracted O-O software, and (2) uses this classification to compare an automated random testing strategy to manual testing, and to furthermore compare testing results to faults detected in the field.

The main results of the comparison are the following. 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 *kind* of faults and not only of the *number* of faults that they find, because only such a comparison can shed light on the particular strengths and weaknesses of each strategy.

Overview. §2 provides the background for random tests of O-O software. §3 presents a classification of faults. The experiments and their discussion are the subject of §4. After putting our work in context in §5, we conclude in §6.

2 Background

This section provides an overview of how AutoTest generates and executes random tests for Eiffel programs. The tool itself is not the focus of this paper and has previously been described elsewhere [11, 12].

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, we must generate an object c of type C and 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, we also have to generate an object r of type R that represents the expected output. We can then execute $c.m(a_1, \ldots, a_n)$, compare the resulting object to 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) can be generated through constructor calls, which themselves are routines and may need arguments; hence, the generation procedure is recursive.

For contract-enabled languages such as Eiffel, JML or Spec#, there is an obvious simplification of the above approach. Rather than generating an object r of type R and checking the effects of executing m on c and the a_i, we can use the routine and class contracts. Upon execution of $c.m(a_1, \ldots, a_n)$, we check if the postcondition of m and the invariant of C are satisfied. In other words, we get an oracle for free, namely in the form of postconditions and invariants. Because it is an abstraction, this oracle is usually not able to predict the precise values that are a result of executing the test. Of course, the contract itself may be incorrect (§3), but this is a general problem with implementing oracles. Furthermore, we have to satisfy the respective precondition whenever a routine (including a constructor) is called.

When generating the input objects for each routine, it is reasonable not to create a new object each time with a constructor call, but also to keep a pool of available objects. This is because constructor calls tend to generate rather "simple" objects. As a consequence, AutoTest retains objects that have been created and stores them in the pool. Whenever it needs an object as target or argument for a call to a routine under test, it randomly picks one of the available objects from the pool. When the routine call is done executing, the object is returned to the pool. With a preset frequency, AutoTest also generates a new object, which it adds to the pool. In this way, the objects in the pool are likely to get modified by the routines under test. Over time, this leads to objects or object structures that are intuitively interesting for testing because they are more representative of objects created and manipulated during typical runs than freshly created objects alone. When values of primitive types are needed as inputs, AutoTest either selects

them from a predefined set of values which includes limit values, or randomly selects one of the possible values for that type. This use of limit values makes the strategy not purely random, so we use the name "random testing+" to refer to it.

Test case generation then means to generate Eiffel code that creates objects and executes their operations – the routines under test. By running this code in an interpreter, contract violations and other uncaught exceptions are logged. These violations are the failures that AutoTest can detect.

In this paper, we adopt the following terminology. A *failure* is an observable difference between expected and actual output. 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. A *fault* is the reason for the error and includes incorrect implementations and invalid specifications, i.e., specifications inconsistent with the actual requirements.

3 Classifications of faults

Two dimensions characterize a fault in programming languages with support for embedding executable specifications. The first is the fault's *location*: whether it occurs in the specification or in the implementation. The second is its *cause*, the real underlying issue. The following paragraphs discuss both dimensions and introduce the resulting fault categories. The classification is not domain-specific. Although, as discussed in §5, a multitude of other fault classification schemes exists, we are not aware of the existence of any such schemes for contracted code.

3.1 Specification&implementation faults

In contract-equipped programs, the software specification is embedded in the software itself. Among other things, failures are triggered by violations of the contracts. 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 necessitates changing the specification (plus possibly also the implementation) [36]. As an example, consider a routine `deposit` of a class `BANK_ACCOUNT` with an integer argument representing the amount of money to be deposited to 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 fulfill its explicit specification, here its contract. The correction of implementation faults never requires changing the specification. Suppose the class `BANK_ACCOUNT` also contains a routine `add_value` that should add a value, positive or negative, to 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.

Naturally, this classification assumes that we have access to the "intended specification" of the software: the real specification that it should fulfill. 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 has no choice but to rely on subjective evidence such as (1) the comments in the routines under test, (2) the specifications and implementations of other routines in the same class, and (3) the ways in which 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 class is supposed to do.

3.2 Classification of faults by their cause

In practice, several classes of specification and implementation faults tend to recur over and over again. 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 for all faults encountered in our study, we obtained the categories described below. The classification was created with practical applicability in mind and mainly focuses on either the mistake in the programmer's thinking or the mis-used programming mechanism that led to the fault.

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

1. In terms of the *precondition*, we identified the following faults.
   - **Missing non-voidness precondition**: A precondition part specifying that a routine argument, class attribute, or another reference reachable through an argument or attribute should not be void (null) is missing.
   - **Missing min/max-related precondition**: a precondition part specifying that an integer argument, class attribute, or another integer reachable through an argument or attribute should have a certain value related to the minimum/maximum possible value for integers is missing.

- **Missing other precondition part**: a precondition is under-specified in another way than the previously mentioned 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 with the most current ones using a disjunction. This fault category refers to faults that appear because of this language mechanism.
- **Precondition too restrictive**: the precondition of a routine is stronger than it should be.

2. In terms of the *postcondition*, the faults are **wrong postcondition** (the postcondition of a routine is incorrect) and **missing postcondition** (the postcondition of a routine was omitted).

3. In terms of *invariants*, we identified only one fault. This is the **missing invariant clause**: a part of a class invariant is missing.

4. In terms of `check`[1] assertions, one fault was identified, namely the **wrong check assertion**: the routine contains a `check` condition that does not hold.

5. Finally, the following faults apply to *all contracts*.

   - **Faulty specification supplier**: a routine called from 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.

The categories in this classification have various degrees of granularity. The reason is that the classification was driven by the actual faults found by AutoTest, by manual testers, and by users of the software. The categories thus emerged by identifying recurring patterns in existing faults, rather than trying to fit faults into predefined, fixed categories. Where such patterns could not be found, the categories are rather coarse-grained.

[1] 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.

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 fulfill 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.
- **Case not treated**: the implementation does not treat one of the cases that can appear, typically in an `if` branch.
- **Catcall**: due to the implementation of type covariance in Eiffel, the compiler cannot detect 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 the expected one.
- **Infinite loop**: executing the routine can trigger an infinite loop, due to a fault in the loop exit condition.

Three of the above categories are specific to the Eiffel language and would not be directly applicable to languages which 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 Experimental results

This section first describes the artifacts examined in the experiment and how the experiment was conducted. Then it

presents the results of the experiment 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 of this paper and one class taken almost verbatim from the EiffelBase library. We then summarize the results, discuss the most important findings, and conclude with a presentation of the threats to the validity of generalizations of the results.

4.1 Experiment

To see how random testing+ performs, we ran AutoTest on classes from the EiffelBase library: the most widely used Eiffel library, containing classes implementing various data structures and other common facilities. Overall, we randomly tested 39 classes from the 5.6 version of the library and found a total of 165 faults in them.

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

To determine how manual testing compares to random testing+, we organized a competition for students of computer science at ETH Zurich. 13 students participated in the competition. They were given 3 classes to test. The task was to find as many faults as possible in these 3 classes in 2 hours. Two of the classes were written by us (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 1 shows some code metrics for these 3 classes: number of lines of code (LOC), number of lines of contract code (LOCC), and number of routines. We intentionally chose one class that was significantly larger and more complex than the others 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 had at least a few lectures on the topic. 9 out of the 13 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 thrown exception.

Table 1. Classes tested manually

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

Table 2. Random Testing+ vs. User Reports

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

4.2 Random testing+ vs. user reports

Table 2 shows the distribution of specification and implementation faults (1) found by random testing+ (labeled "AutoTest" in the table) 39 classes from the EiffelBase library and (2) recorded in bug reports from the users for the same library. Note that the results in this table refer to more classes tested with AutoTest than for which there are user reports. This is justified by the fact that even if there are no user reports on a specific class, this does not mean that the class was not used in the field.

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 3 presents a finer-grained 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 §3. This detailed comparison sheds more light on differences between faults reported by users and those found by automated testing, and exposes strengths and weaknesses of both approaches. One difference that stands out refers to 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 or simply that users do not consider them to be worth reporting. Furthermore, it is possible that users use their intuition of the range of acceptable inputs for a routine rather than using that routine's precondition, and their intuition corresponds to the real specification, not to the erroneous one provided in the contracts.

A further difference results from AutoTest's ability to detect faults from the categories "faulty specification supplier" and "faulty implementation supplier." In other words, AutoTest can report that certain routines do not work properly

Table 3. Random Testing+ vs. User Reports: Specification and Implementation Faults

Cause	Id	Number of faults		Percentage of faults	
		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%*

because they depend on other faulty routines. Users never report this kind of faults: they only indicate the routine that contains the fault, without mentioning other routines that also do 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: overly 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 category. It is not surprising that AutoTest has trouble detecting such faults. Firstly, if AutoTest tries to call a routine with an overly strong precondition and does not fulfill this precondition, the testing engine will simply classify the test case as invalid and try again to satisfy the routine's precondition by using other inputs. Secondly, 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. Thirdly, 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. It is of course possible to find empty routine bodies statically, but it is not possible to decide if this is indeed a fault. Note that in these cases, the overall number of detected faults is rather low, which suggests special care in generalizing these findings.

We also ran AutoTest exclusively on the classes for which users reported faults to see if it would find those faults; three of the classes with reported faults are excluded from the analysis because of technical limitations of AutoTest (it cannot test expanded and built-in classes). When run on each class in 10 sessions of 3 minutes, AutoTest found a total of 268 faults[2]. 4 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

[2]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 faults in the software.

Table 4. 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%)	

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." It is important to note though that longer testing time might have produced different results.

4.3 Random testing+ vs. manual testing

To investigate how AutoTest performs when compared to manual testers (the students participating in the competition), we ran AutoTest on the 3 classes that were tested manually. The tool tested each class in 30 sessions of 3 minutes, where each session used a different seed for the pseudo-random number generator. Table 4 shows a summary of the results. It displays a categorization of the fault according to the classification scheme used in this paper (the category ids are used here; they can be looked up in Table 3), 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 2 classes. We conjecture that, because of its size, students tested this class less thoroughly than the others. This highlights one of the (possibly obvious) 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.

4.4 Summary and consequences

Three main observations emerge from the preceding analysis. First, 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.

Second, 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.

Third, in a comparison between automated and manual testing (i.e., not taking into consideration bug reports), AutoTest detects almost all problems also detected by humans, plus a few other problems. For model-based testing, this confirms the findings of an earlier study [36] and speaks strongly in favor 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.

4.5 Discussion

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 our study). Two questions naturally arise. One, are there more faults in contracts than in implementations, i.e., do the results obtained with AutoTest reflect the actual distribution of faults? Two, is it interesting at all to find faults in contracts, knowing that contract checking is usually disabled in production code?

We do not know the answer to the first question. We cannot deduce from our results that there are indeed more problems in specifications than in implementations. The only thing we can 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 scheduled to go away as the new versions of Eiffel, starting with 6.2 (Spring 2008), implement the "attached type" mechanism [31] which removes the problem by making non-voidness part of the type system and catches violations at compile time rather than run time.

As to the question 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. We do not consider this to be too strong an assumption because it has been shown [9] 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 previous work [12] we have shown that random testing+ is predictable in the *number* of faults that it finds, but not in the *kind* of faults that 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 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.

4.6 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, of the analyzed user bug reports, and of the group of human testers participating in the study. In future work we intend to expand this study to larger and more diverse code bases. Nevertheless, we consider the results presented here to be a first major step in the direction of comparing random and manual testing and user bug reports by the type of faults they reveal.

As explained in §4.1, we only had access to bug reports submitted by users for the EiffelBase library. Naturally, these are not all the faults found in field use of the library, but only the ones that users took the time to report. It is interesting to note that 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 7 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.

As we could not perform the study with professional testers, we used bachelor and master students of Computer Science who were motivated with the prices of the competition to find as many faults as possible. In a questionnaire they filled in after the competition, 4 of the students declared themselves to be proficient programmers and 9 estimated they had "basic programming experience." 7 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. Furthermore, as mentioned in §4.1, 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 actual faults occurring in real software, so they were created as instances of various categories described in §3, but naturally the fact that they were seeded in code written by one of the authors introduces a bias in the results. All these aspects limit the generality of our conclusions.

A further threat to the generalization of our results stems from 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 OO programs (e.g., [13]). In particular, we make use of extreme values to initialize the object pool (§2). While void objects are rather likely to occur in practice, extreme integer values are not. In other words, as mentioned earlier, the approach is not entirely "random."

As noted, compile-time removal of void-related errors will affect the results, for Eiffel and other languages that have the equivalent of an "attached type" mechanism (for instance Spec# [4]).

Another source of uncertainty is the assignment of defects to a classification. Finding a consistent assignment among several experts is difficult [25]. In our study, one person was assigned to this task. We hence believe that we

have consistent results. Nevertheless, a similar experiment with a different person might exhibit different results.

Finally, the programming language used in the study, Eiffel, also influenced the results. As explained in §4.5, a few of the fault categories stem from the language mechanisms that are mis-used 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.

5 Related work

Many fault classification models have been proposed in the past [10, 1, 30, 21, 6, 35]. This includes the Orthogonal Defect Classification (ODC) [10] whose main point is to combine two different classifications, defect types and defect triggers. In a sense our classification is an ODC in itself but our classification of defect types is finer while the defect location is simpler than defect triggers. The IEEE classification [1] 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 our case using such categories would not have helped because they do not reflect the particular constructs of contract-enabled languages. Lutz [30] describes a safety checklist that defines categories of possible errors in safety-critical embedded systems. The classification probably most similar to ours is the one used by Basili et al. [5, 28, 39], organized in two dimensions: whether the fault is an omission or a commission fault, and to which of 6 possible types it belongs. Our classification takes into account specifications (contracts) and is more fine-grained.

Bug patterns (e.g., [2, 26]) are also related to our work. Allen [2] defined 14 types of defects in Java programs. Our approach has to cope with different constructs and thus defines categories adapted to Eiffel programs, taking into account contracts and multiple inheritance.

Several studies compare different testing strategies by evaluating their respective failure-triggering capabilities. Random tests have been compared to partition testing both theoretically and empirically [22, 38, 15, 23], and the effectiveness of model-based testing has been studied [36, 24, 7]. These studies do not take into consideration a classification of faults which, among other things, led us to conduct the experiment presented here.

Numerous other studies have compared structural and functional testing strategies as well as code reading (among others, [18, 5, 28, 39]). None of these studies compares manual testing to automated techniques. Our study compares random testing+ with manual testing and user bug reports; as far as we know, this is the first time that these three methods of identifying software faults are correlated. Like most of the earlier comparative studies, the results highlight the complementarity of different techniques.

6 Conclusions

One of the main goals of this work is to understand if different ways of detecting faults detect different kinds of faults. The question is of high practical importance: which testing strategy should be applied under which circumstances? A further motivation was the conjecture that a reason for the inconclusiveness of earlier comparative studies is that the number of detected faults alone is too strong an abstraction for comparing testing strategies.

More specifically, in this paper we examined the kind of faults that random testing+ finds, and the question whether and how these differ from faults found by human testers and by users of the software. The experiments presented here suggest that these three 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. The experiments presented here as well as in earlier work [11, 12] show that the tool indeed finds a high number of faults in little time. 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 latter experiment also proved that AutoTest finds faults that testers miss. The conclusion is that random tests+ should be used alongside with manual tests. Given earlier results on comparing different QA strategies, this is not surprising, but we are not aware of any systematic studies that showed this for random testing. We discussed threats to generalizing these results in §4.6.

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

Our analysis is based on a classification of faults that is not specific to one particular application domain. In terms of specification faults, it caters to typical problems with pre- and postconditions and invariants that are too weak, e.g., do not take into account extreme values, or are too strong. In terms of implementation faults, there are a few general problems such as missing cases or infinite loops, and some problems that relate to the idiosyncrasies of the Eiffel language. We do not claim that our classification is complete or the only possible one; it is the result of analyzing the faults that we encountered. We believe that this classification, or some variant of it, can be used for future experiments on comparing strategies for finding faults.

Acknowledgments. We thank Raluca Borca, on whose work a large part of the study of random testing is based. We are also thankful to Andreas Leitner for numerous discussions and insightful observations on this work.

References

[1] IEEE standard classification for software anomalies. *IEEE Std 1044-1993*, 2 Jun 1994.
[2] E. Allen. *Bug Patterns in Java*. APress L. P., 2002.
[3] S. Artzi, M. Ernst, K. A, C. Pacheco, and J. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *Proc. 1st Workshop on Model-Based Testing and Object-Oriented Systems*, 2006.
[4] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In *Proc. CASSIS*, 2004. Springer LNCS 3362.
[5] V. Basili and R. Selby. Comparing the effectiveness of software testing strategies. *IEEE TSE*, 13(12):1278–1296, 1987.
[6] B. Beizer. *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990.
[7] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *Software-Practice & Experience*, 34(10):915–948, August 2004.
[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. Intl. Symp. on Software Testing and Analysis*, pages 123–133, 2002.
[9] P. Chalin. Are practitioners writing contracts? In *Springer LNCS 4157*, pages 100–113, 2006.
[10] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M.-Y. Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE TSE*, 18(11):943–956, 1992.
[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proc. ISSTA*, pages 84–94, 2007.
[12] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *Proc. ICST*, pages 72–81, 2008.
[13] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.
[14] C. Csallner and Y. Smaragdakis. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. In *Proc. Intl. Symp. on Software Testing and Analysis*, pages 245–254, July 2006.
[15] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE TSE*, SE-10(4):438–444, July 1984.
[16] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, August 2000.
[17] P. Frankl and S. Weiss. An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing. *IEEE TSE*, 19(8):774–787, 1993.
[18] P. Frankl, S. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness, 1994.
[19] P. Frankl and E. Weyuker. An Applicable Family of Data Flow Testing Criteria. *IEEE TSE*, 14(10):1483–1498, 1998.
[20] M. Girgis and M. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Proc. IEEE/ACM workshop on software testing*, pages 64–73, July 1986.
[21] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.
[22] W. Gutjahr. Partition testing versus random testing: the influence of uncertainty. *IEEE TSE*, 25(5):661–674, 1999.
[23] D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE TSE*, 16(12):1402–1411, Dec. 1990.
[24] M. Heimdahl, D. George, and R. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proc. 8th IEEE High Assurance in Systems Engineering Workshop*, February 2004.
[25] K. Henningsson and C. Wohlin. Assuring fault classification agreement – an empirical evaluation. In *Proc. Intl. Symp. on Empirical Software Engineering*, pages 95–104, 2004.
[26] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
[27] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. ICSE*, pages 191–200, 1994.
[28] E. Kamsties and C. Lott. An empirical evaluation of three defect-detection techniques. In *Proc. ESEC*, pages 362–383, 1995.
[29] A. Leitner and I. Ciupa. AutoTest. `http://se.inf.ethz.ch/people/leitner/auto_test/`, 2005 - 2007.
[30] R. Lutz. Targeting Safety-Related Errors During Software Requirements Analysis. In *Proc. ACM SIGSOFT FSE*, pages 99–106, 1993.
[31] B. Meyer. Attached types and their application to three open problems of object-oriented programming. In *Proc. ECOOP*, pages 1–32, 2005. Springer LNCS 3586.
[32] C. Oriat. 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.
[33] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. ECOOP*, pages 504–527, 2005.
[34] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, 2007.
[35] J. Ploski, M. Rohr, P. Schwenkenberg, and W. Hasselbring. Research issues in software fault categorization. *SIGSOFT Softw. Eng. Notes*, 32(6):6, 2007.
[36] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proc. ICSE*, pages 392–401, 2005.
[37] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato (New Zealand), April 2006.
[38] E. Weyuker and B. Jeng. Analyzing Partition Testing Strategies. *IEEE TSE*, 17(7):703–711, 1991.
[39] M. Wood, M. Roper, A. Brooks, and J. Miller. Comparing and combining software defect detection techniques: a replicated empirical study. *SIGSOFT Softw. Eng. Notes*, 22(6):262–277, 1997.