Experimental Assessment of Random Testing for Object-Oriented Software

Ilinca Ciupa, Andreas Leitner, Manuel Oriol, Bertrand Meyer
Chair of Software Engineering
Swiss Federal Institute of Technology Zurich
CH-8092 Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

ABSTRACT

Progress in testing requires that we evaluate the effectiveness of testing strategies on the basis of hard experimental evidence, not just intuition or a priori arguments. *Random testing*, the use of randomly generated test data, is an example of a strategy that the literature often deprecates because of such preconceptions. This view is worth revisiting since random testing otherwise offers several attractive properties: simplicity of implementation, speed of execution, absence of human bias.

We performed an intensive experimental analysis of the efficiency of random testing on an existing industrial-grade code base. The use of a large-scale cluster of computers, for a total of 1500 hours of CPU time, allowed a fine-grain analysis of the individual effect of the various parameters involved in the random testing strategy, such as the choice of seed for a random number generator. The results provide insights into the effectiveness of random testing and a number of lessons for testing researchers and practitioners.

Categories and Subject Descriptors

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

General Terms

Measurement, Verification

Keywords

software testing, random testing, experimental evaluation

1. OVERALL GOALS

The research effort invested into software unit testing automation during recent years and the emergence of commercial applications implementing some of the resulting ideas (such as Agitar's Agitator [5] or Parasoft's Jtest [2]) are evidence of the attraction of automated testing solutions.

One approach to fully automated testing is random testing (provided that an automated oracle is also available). In the software testing literature, the random strategy is often considered to be one of the least effective approaches. The advantages that random testing does present (wide practical applicability, ease of implementation and of understanding, execution speed, lack of bias) are considered to be overcome by its disadvantages. However, what stands behind this claim often seems to be intuition, rather than experimental evidence.

A number of years ago Hamlet [18] already pointed out that many of the assumptions behind popular testing strategies, and many ideas that seem intuitively to increase testing effectiveness, have not been backed by experimental correlation with software quality.

Serious advances in software testing require a sound and credible experimental basis to assess the effectiveness of proposed techniques. As part of a general effort to help establish such a basis, we have investigated the effectiveness of random testing at finding bugs in a significant code base with two distinctive properties: it is extensively used in production applications and it contains a number of bugs, which can be found through automatic testing. This makes it possible to assess testing strategies objectively, by measuring how many of these bugs they find and how fast. To allow individual assessment of the many parameters involved in defining such a strategy, we performed large-scale experiments with the help of a cluster consisting of 32 dual-core machines running in parallel. The total automatic testing effort resulted in 1875 test session results for each of the 8 classes under test, over a total CPU time of 1500 hours.

A study of this size was only possible in the presence of an automated oracle. In our approach, this oracle is provided by executable specification embedded into the source code in the form of contracts as described in the Design by Contract software development methodology [21]. We used the AutoTest tool ([10], [19] describe previous versions of it), which takes advantage of contracts to uncover bugs automatically.

Section 2 describes the setup of the experiments, including some details on AutoTest, and Section 3 the results; Section 4 presents a discussion of the experiment. Section 5 reviews related work and Section 6 draws some conclusions.

Here as a preview is a *summary of the main results*:

- The number of found bugs has a surprisingly high increase in the first few minutes of testing.
- The seed used for the random testing algorithm can make an important difference in the number of bugs found over the same timeout. For example, in the case of one class under test, 5 bugs were found for one seed and 23 for another one over a timeout of 30 minutes, all other conditions being the same.

ISSTA'07, July 9–12, 2007, London, England, United Kingdom.

- The version of the random testing algorithm that works best for a class for a testing timeout as small as 2 minutes will also deliver the best results for higher timeouts (such as 30 minutes).

- This version is not the same for all classes, but one can identify a solution that produces optimal results for a specified set of tested classes.

These conclusions are of course to be interpreted in light of our specific setup and assumptions, which are made explicit in the following sections together with the details of the results.

In addition to the results summarized above, this paper also makes the following contributions:

- *Suggestions for testing practitioners and tool builders.* Based on the results of our study, we provide guidelines for choosing which algorithm for random testing should be used depending on the test scope and duration.

- *Basis for other case studies.* The results we show here can be used to open the way for a series of comprehensive studies of the performances of various automated testing strategies. In particular, the algorithms identified here to produce the best results for random testing should be used as the basis for comparison against any other strategy.

2. TEST BED

2.1 Framework

AutoTest, the tool we used for running the experiment, achieves push-button testing of Eiffel applications equipped with contracts. It is implemented as a framework, so that various strategies for input value generation can easily be plugged in. For the experiment described in this paper we used a configurable random strategy.

Input generation

Random input generation for object-oriented applications can be performed in one of two ways:

- In a *constructive* manner, by calling a constructor of the targeted class and then, optionally, other methods of the class in order to change the state of the newly created object.

- In a *brute force* manner, by allocating a new object and then setting its fields directly (technique applied by the Korat tool [6], for instance).

The second approach has the disadvantage that objects created in this way can violate their class invariant. If this is the case, the effort for creating the object has been a waste and a new attempt must be made. (Invariant violations can also be triggered through the constructive approach, but, if that is the case, a bug has been found in the system under test.) Also, it can be the case that objects created in this way could never be created by executions of the software itself. For these reasons, AutoTest implements the first approach.

Any such constructive approach to generating objects (test inputs) must answer the following questions:

- Are objects built simply by constructor calls or are other methods of the class called after the constructor, in an attempt to bring the object to a more interesting state?

- How often are such diversification operations performed?

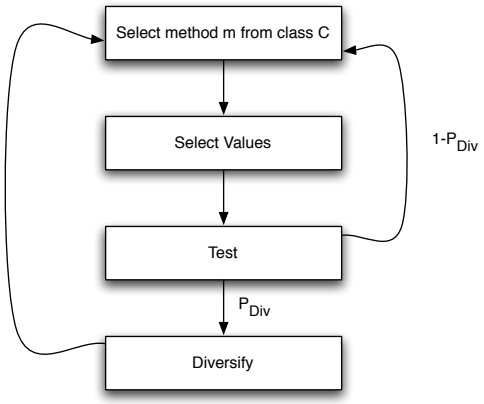

Figure 1: General algorithm

- Can objects used in tests be kept and re-used in later tests?

- How are values for primitive types generated?

With AutoTest, answers to these questions take the form of explicit parameters provided to the input generation algorithm.

AutoTest keeps a pool of objects available for testing. All objects created as test inputs are stored in this pool, then returned to the pool once they have been used in tests. The algorithm for input generation proceeds in the following manner, given a method `m` of a class `C` currently under test. To test `m`, a target object and arguments (if `m` takes any) are needed. With some probability P_{GenNew}, the algorithm either creates new instances for the target object and arguments or uses existing instances (taken from the pool). If the decision is to create new instances, then AutoTest calls a randomly chosen constructor of the corresponding class (or, if the class is abstract, of its closest non-abstract descendant). If this constructor takes arguments, the same algorithm is applied recursively. The input generation algorithm treats basic types[1] differently: for an argument declared of a primitive type, with some probability $P_{GenBasicRand}$, a value will be chosen randomly either out of the set of all values possible for that type or out of a set of predefined, special values. These predefined values are assumed to have a high bug-revealing rate when used as inputs. For instance, for type INTEGER, these values include the minimum and maximum possible values, 0, 1, -1, etc. Figures 1 and 2 depict an overview of the use of these probabilities in the input generation algorithm.

Keeping a pool of objects which can be reused in tests raises the question of whether an attempt should be made to bring these existing objects to more interesting states as would occur during the actual execution of a program (for example with a list class, not just a state occurring after creation but one resulting from many insertions and deletions). To provide for this, we introduce the probability P_{Div} which indicates how often, after running a test case, a *diversification* operation is performed. Such a diversification operation consists of calling a procedure (a method with no return value) on an object selected randomly from the pool.

Supplying different values for these three probabilities (P_{GenNew}, $P_{GenBasicRand}$, P_{Div}) changes the behavior of the input generation algorithm.

[1]Basic types are also called "primitive" and among these types in Eiffel are INTEGER, REAL, DOUBLE, CHARACTER, and BOOLEAN.

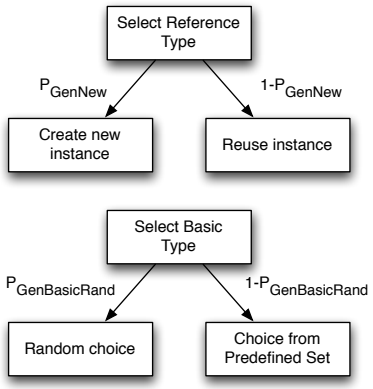

Figure 2: Value Selections

Other tools which use a constructive approach to random input generation also rely on calling sequences of constructors and other methods to create input data. Eclat [23], very much like AutoTest, stores objects in a pool and uses existing values to call constructors and methods which create new values. However, after this initial generation phase, it applies heuristics to classify and select which of the available inputs it will use in tests. AutoTest performs no such selection, because it wants to implement a purely random strategy. JCrasher [12] builds sequences of constructor and method calls starting from a method under test and systematically building required input objects, by calling either constructors or methods returning objects of the desired type.

Test execution

AutoTest uses a two-process model for executing the tests: the *master* process implements the actual testing strategy; the *slave* process is responsible for the test execution. The slave, an interpreter, gets simple commands (object creation, method call, etc.) from the master and can only execute such instructions and return the results. This separation of the testing activity in two processes has the advantage of robustness: if test execution triggers a failure in the slave from which the process cannot recover, the interpreter will shut it down and then restart it where testing was interrupted. The entire testing process does not have to be restarted from the beginning and, if the same failure keeps occurring, testing of that method can be aborted so the rest of the test scope can still be explored.

Automated oracle

AutoTest uses contracts (method pre- and postconditions and class invariants) present in the code as an automated oracle. In Eiffel these contracts are boolean expressions (with the addition of the `old` keyword which can only appear in postconditions and is used to refer to the value of a variable before the execution of the method), so they are easy to learn and use even for beginner programmers. As these contracts contain the specification of the software and can be evaluated at run time, AutoTest checks them during testing and reports any contract violation (with the exception of cases in which a test input directly violates the precondition of the method under test).

An important distinction must be noted. The information that we look for in our experiment is not the number of *failures* (executions leading to outcomes different from the expected ones), but that of software *faults* (problems in the code that can trigger failures). Since a single fault can trigger multiple failures, we consider that two failures expose the same fault if they are triggered at the same line in the code and manifest themselves through the same type of exception. (In Eiffel, contract violations are also exceptions, but, for clarity, for the rest of this paper we will refer to them separately.) Hence, under this convention, the measures that we provide in our results always represent the number of found faults, not failures. Such faults are simply called "bugs" below, although this is not a rigorous terminology.

2.2 Experimental setup

The experiment was run using the ISE Eiffel compiler version 5.6 on 32 identical machines, each having a Dual Core Pentium III at 1 GHz and 1 Gb RAM, running Fedora Core 1.

We chose the classes to test in the experiment so that they come from different sources and have varying purposes, sizes, and complexity:

- Classes from the data structures library of Eiffel, used by most projects written in this language (EiffelBase 5.6 [1]): STRING, PRIMES, BOUNDED_STACK, HASH_TABLE.
- Classes written by students of the Introduction to Programming course at ETH Zurich for an assignment: FRACTION1, FRACTION2.
- Classes mutated to exhibit some common bugs found in object-oriented applications: UTILS, BANK_ACCOUNT.

The last four classes are available at http://se.inf.ethz.ch/people/ciupa/test_results. The others are available as part of the EiffelBase library version 5.6 [1]. The classes from the EiffelBase library and those written by students were not modified in any way for this experiment.

Table 1 shows various data about the classes under test: total number of lines of code, number of lines of contract code, number of methods (including those inherited), number of parent classes (also those that the class indirectly inherits from). In Eiffel all classes inherit implicitly from class ANY (similarly to the case of Java and class Object), so every class has at least one parent class.

We tested each of the classes for 30 minutes, for three different seeds for the pseudo-random number generator, for all combinations of the following values for each parameter to the input generation algorithm:

- P_{GenNew} (the probability of creating new objects as inputs rather than using existing ones) $\in \{0; 0.25; 0.5; 0.75; 1\}$
- P_{Div} (the probability of calling a procedure on an object chosen randomly from the pool after running each test case) $\in \{0; 0.25; 0.5; 0.75; 1\}$
- $P_{GenBasicRand}$ (the probability of generating values for basic types randomly rather than selecting them from a fixed predefined set of values) $\in \{0; 0.25; 0.5; 0.75; 1\}$

Thus, we ran AutoTest for each of these classes for 30 minutes, for every combination of the 3 seed values, 5 values for P_{GenNew}, 5 values for P_{Div}, and 5 values for $P_{GenBasicRand}$. So there were 3 * 5 * 5 * 5 = 375 tests run per class for 30 minutes each, amounting to a total test time of 90000 minutes or 1500 hours.

We then parsed the saved test logs to get the results for testing for 1, 2, 5, 10, and 30 minutes. (This approach is valid since AutoTest tests methods in the scope in a fair manner, by selecting at each step the method that has been tested the least up to the current moment. This means that the timeout that the tool is given does not influence how it selects which method to test at any time.)

Table 1: Properties of the classes under test

Class	Total lines of code	Lines of contract code	Number of methods	Number of parent classes
STRING	2600	283	175	7
PRIMES	262	52	75	1
BOUNDED_STACK	249	44	66	2
HASH_TABLE	1416	156	135	3
FRACTION1	152	36	44	1
FRACTION2	180	32	45	1
UTILS	54	34	32	1
BANK_ACCOUNT	74	43	35	1

Hence, for each combination of class, seed, timeout, and probability values, we get the total number of found "bugs" (in the above sense) and the number of these bugs which were found due to contract violations and due to other exceptions, respectively. Since there are 5 timeout values and 375 tests/class, this method produced 5 * 375 = 1875 test session results per class.

This paper only reproduces part of the raw data. The results and conclusions are based on the entire raw data, available at http://se.inf.ethz.ch/people/ciupa/test_results.

The criterion we used for evaluating the efficiency of the examined strategies is *the number of bugs found in a set time*. Although several other criteria are commonly used to evaluate testing strategies, we consider this criterion to be the most useful, since the main purpose of unit testing is to find bugs in the modules being tested.

3. RESULTS

This section analyzes the results with respect to the questions stated as the goals of the experiment.

How does the number of found bugs evolve over time?

To determine how the number of found bugs evolves with the timeout we look at the highest number of bugs (averaged over the three seeds) found for each timeout for every class. For the classes that were tested, the evolution was inversely proportional to the elapsed time: the best fitting that we could find was against a function $f(x) = a/x + b$. Table 2 shows, for each class under test, the parameters characterizing the fitting of the evolution of the number of found bugs over time against this function with 95% confidence level. The parameters quantifying the goodness of fit are:

- SSE (sum of squared errors): measures the total deviation of the response values from the fit. A value closer to 0 indicates that the fit will be more useful for prediction.
- R-square: measures how successful the fit is in explaining the variation of the data. It can take values between 0 and 1, with a value closer to 1 indicating that a greater proportion of variance is accounted for by the model.
- RMSE (root mean squared error): an estimate of the standard deviation of the random component in the data. An RMSE value closer to 0 indicates a fit that is more useful for prediction.

Figures 3, 4, and 5 illustrate the results of the curve fitting for classes STRING, PRIMES, and HASH_TABLE respectively. They show both the best fitting curves (as given in Table 2) and the actual data obtained in the experiment.

Table 2: Parameters characterizing the fitting of the evolution of the number of found bugs over time against the function $f(x) = a/x + b$.

Class name	a	b	SSE	R-square	RMSE
BANK_ ACCOUNT	-1.25	2.58	0.03	0.96	0.10
BOUNDED_ STACK	-4.20	11.6	0.11	0.98	0.23
FRACTION1	-0.45	3.14	0.01	0.86	0.10
FRACTION2	-2.19	2.93	0.45	0.86	0.38
HASH_TABLE	-14.48	19.57	9.64	0.93	1.79
PRIMES	-4.8	6.96	0.39	0.97	0.36
STRING	-10.37	11.47	1.24	0.98	0.64
UTILS	-3.16	3.82	0.15	0.97	0.22

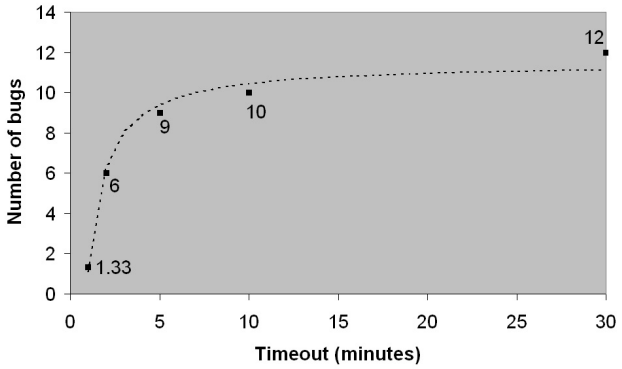

Figure 3: Evolution of the number of found bugs over time for class STRING

How much does the seed influence the number of found bugs?

As stated, the experiment ran each combination of probabilities, for each class, for each timeout, and for 3 different seeds. The results show that the seed has a high influence on the number of found bugs, going as far as, for class HASH_TABLE, finding 5 bugs for one value of the seed and 23 for another one (for the same timeout of 30 minutes and the same combination of probability values). Table 3 shows for each class the number of bugs found for different seeds for the same combination of probability values and for the same timeout and which have the highest difference between the maximum and minimum number of found bugs. The table also shows the timeouts for which the given number of bugs was found.

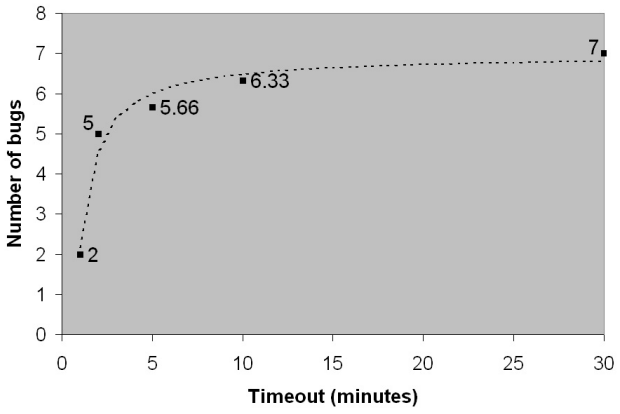

Figure 4: Evolution of the number of found bugs over time for class PRIMES

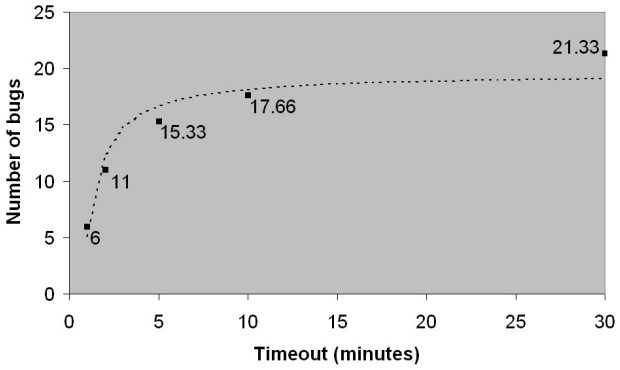

Figure 5: Evolution of the number of found bugs over time for class HASH_TABLE

The varying values for these timeouts show no correlation between the maximum difference that the seed can make in the number of found bugs and the testing timeout.

The values shown in Table 3 are extreme cases. The distribution of all differences in the number of bugs found for the different seed values for a certain class is also interesting. Figure 6 shows this distribution for class HASH_TABLE. The figure shows the number of occurrences for every difference in the number of bugs found for every combination of probabilities and timeout value for the different seeds. In other words, it shows how often each difference in number of found bugs occurs, illustrating how important the influence of the seed can be (all other things being equal) and how often such an influence occurs. Figure 7 shows the same information for class BOUNDED_STACK.

Although these results indicate how important the influence of the seed can be, this should not lead one to believe that a certain seed value constantly delivers better results than another one. Instead, these results indicate that random testing needs to be performed several times, with various seed values, to compensate for the high variability.

Table 3: Minimum and maximum number of bugs found maximizing the difference obtained for using different seeds for the same timeout and combination of probabilities

Class name	Max bugs	Min bugs	Timeout
BANK_ACCOUNT	2	0	1, 2, 5, 10
BOUNDED_STACK	9	2	2
	10	3	5
FRACTION1	3	0	1
FRACTION2	3	0	5, 10
HASH_TABLE	23	5	30
PRIMES	7	0	30
STRING	7	2	2
	12	7	30
UTILS	3	0	2

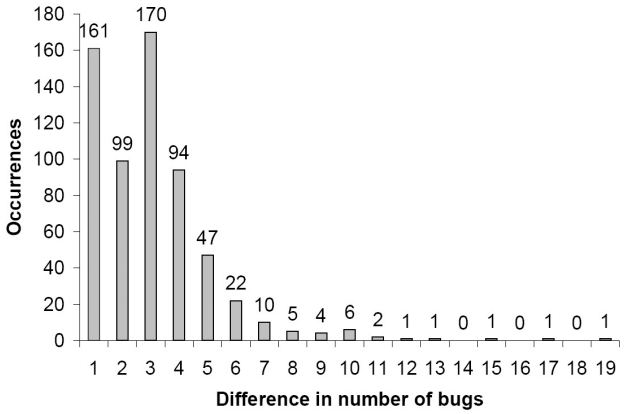

Figure 6: Distribution of the difference in number of found bugs caused by the used seed for class HASH_TABLE

How much do the values of the probabilities influence the number of found bugs for every timeout?

Table 4 shows the minimum and maximum number of bugs found (averaged over the three seeds) for every timeout using all combinations of probability values for each class.

These results show that the *minimum* number of bugs found stays constant or increases very little with the increase of the timeout. In other words, for each timeout, there exist several combinations of probabilities which perform surprisingly badly compared to others. For all classes under test (with the exception of PRIMES) a value of 0 for the probability of generating new objects delivered bad results. A value of 1 for the probability of generating basic values randomly had the same effect.

Classes HASH_TABLE, STRING, and BOUNDED_STACK especially show a high difference between the maximum and minimum numbers of bugs found for every timeout. This shows that the performance of the random testing algorithm can vary widely with the combination of probabilities that is chosen.

Which version of the random generation algorithm maximizes the number of found bugs?

The goal of this analysis is to find the combination of probability values that maximizes the number of found bugs, first over all tested

Table 4: Minimum and maximum number of bugs found for each timeout

Time-out	STRING Min bugs	STRING Max bugs	UTILS Min bugs	UTILS Max bugs	PRIMES Min bugs	PRIMES Max bugs	BANK_ACCOUNT Min bugs	BANK_ACCOUNT Max bugs	BOUNDED_STACK Min bugs	BOUNDED_STACK Max bugs
1	0.00	1.33	0.00	0.66	0.33	2.00	0.00	1.33	0.66	7.33
2	0.00	6.00	0.00	2.33	3.00	5.00	0.00	2.00	1.00	9.66
5	0.00	9.00	0.00	3.00	3.33	5.66	0.00	2.33	1.00	11.00
10	0.00	10.00	0.00	3.33	3.33	6.33	0.00	2.33	1.00	11.00
30	0.00	12.00	0.00	4.00	3.33	7.00	0.00	2.66	1.00	11.00

Time-out	FRACTION1 Min bugs	FRACTION1 Max bugs	FRACTION2 Min bugs	FRACTION2 Max bugs	HASH_TABLE Min bugs	HASH_TABLE Max bugs
1	0.00	2.66	0.00	1.00	1.00	6.00
2	2.00	3.00	0.00	1.33	1.00	11.00
5	2.00	3.00	0.00	2.33	1.00	15.33
10	2.00	3.00	0.00	3.00	1.00	17.66
30	2.00	3.00	0.00	3.00	1.00	21.33

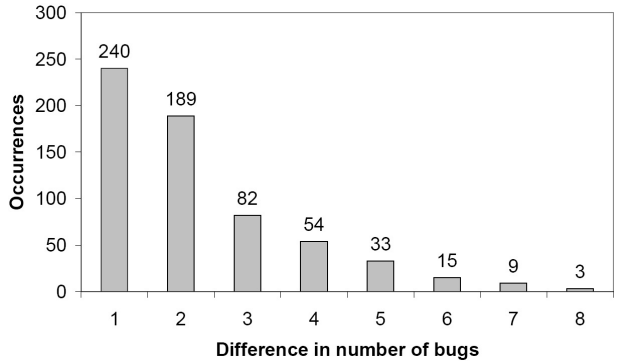

Figure 7: Distribution of the difference in number of found bugs caused by the used seed for class BOUNDED_STACK

classes and over all timeouts, and then individually per class and timeout (1, 2, 5, 10, and 30 minutes). Since the seed influences the results, the results are averaged over the 3 seed values.

The best combination of probabilities averaged over all classes and timeouts is C_0 such that $P_{GenNew0} = 0.25$, $P_{Div0} = 0.5$, and $P_{GenBasicRand0} = 0.25$. With this combination of probabilities, the *average* percent of bugs that is lost by timeout and by class compared to the highest number of bugs that could be found (by the optimal combination of probability values for that specific class and timeout) is 23%. When trying to bound the percent of bugs that could not be found by C_0, only two combinations give *bounded* values: $P_{GenNew1} = 1$, $P_{Div1} = 0.25$, $P_{GenBasicRand1} = 0.25$ ignores 70% of the bugs at most, and $P_{GenNew2} = 1$, $P_{Div2} = 0.75$, $P_{GenBasicRand2} = 0$ ignores 75% of the bugs at most.

If the low timeout values (1 and 2 minutes) are excluded from this calculation, then combination C_0 does not find at most 44% of the bugs. This is not significantly different from the best value: 43%. The most likely explanation is that the low timeout values introduce a high level of noise in the equations due to the low number of tests performed in each series.

Another analysis groups results by classes and looks for tendencies over each of the classes. Table 5 gives a more detailed view of the results. For each class and for each timeout of 2, 5, 10, and 30 minutes, the table shows values for P_{GenNew} (abbreviated in the table as P_{New}) and $P_{GenBasicRand}$ (abbreviated in the table as P_{Basic}) that uncover the highest number of bugs. When there is more than one value, several values uncover the same number of bugs. If the difference between the highest and second highest numbers of bugs is very low, the table also shows the second highest number of bugs. The question marks stand for inconclusive results, that is cases where there were several values for the probabilities which uncovered the same maximum number of bugs, and there was no clearly predominant value for the probability in question. The probability of diversifying is not shown in the table because clear correlation could not be made between its value and the number of bugs. Results for classes FRACTION1 and FRACTION2 were also unclear. The issue with these classes was that the total number of bugs is small (3) and a minimal variation impacts greatly on the tendency.

The results show that the most effective probability values differ from class to class, but, in most cases, they either change very little or not at all with the timeout for a particular class. In other words, for a certain class, the same combinations provide the best results regardless of the timeout of testing.

According to the results in Table 5, a value of 0.25 for P_{GenNew} seems generally to deliver good performance, confirming the result explained above. Exceptions from this rule are classes BOUNDED_STACK, PRIMES, and UTILS. A likely explanation for the different behavior of the last two classes is that very little of their behavior is dependent on their state, so, if they contain any bugs, these bugs will manifest themselves on newly created objects too, and not only on objects in a certain state.

Low values for $P_{GenBasicRand}$ (0, 0.25) also seem to deliver the best results in most cases. Again, classes BOUNDED_STACK and PRIMES have different behavior: in these classes and in class HASH_TABLE, the most bugs are uncovered for $P_{GenBasicRand}$ = 0.75 or 1. In the case of class PRIMES, the most obvious reason is its very nature: a class implementing the concept of prime numbers is best tested with random values, not with values chosen from a limited set, which have no relation to the characteristics of the class.

As a general conclusion, the combination of factors C_0 gives the best overall result but the process can be fine-tuned depending on the classes that are tested. This fine-tuning is not dependent on the timeout value chosen. However, if time permits, one should run random testing several times with different values for the parame-

ters, because, even though a certain combination of parameters may find fewer bugs than another, it may find *different* bugs.

Are more bugs found due to contract violations or due to other exceptions being thrown?

The Design by Contract software development methodology recommends that the contracts be written at the same time as (or even before) the implementation. Eiffel programmers generally follow this practice, but the contracts that they write are most often weaker (especially in the case of postconditions and class invariants) than the intended specification of the software. When contracts are used as oracles in testing, any condition that is not expressed in them cannot be checked, so bugs might be missed. For this reason we consider uncaught exceptions also to signal bugs. But what contribution does each of these two factors have to the total number of found bugs?

Figure 8 shows the evolution over time of the number of bugs found through contract violations and that of the number of bugs found through other exceptions being thrown for class STRING. The values shown on the graph are obtained by averaging over the numbers of bugs found for every timeout by all versions of the random algorithm. For most other classes this evolution is similar. One concludes that over time the proportion of bugs found through contract violations becomes much higher than that of bugs found through other thrown exceptions. For short timeouts (1 or 2 minutes) the situation is reversed.

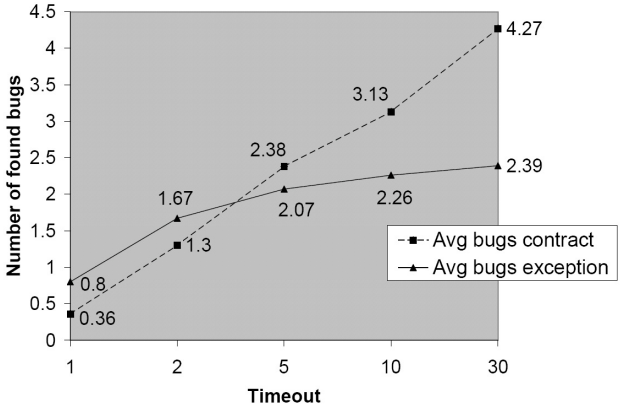

Figure 8: Evolution over time of the number of bugs found through contract violations and through other exceptions for class STRING

Extreme cases are those of classes BOUNDED_STACK, BANK_ACCOUNT, and PRIMES. Figure 9 shows the evolution over time of the number of bugs found through contract violations and that of the number of bugs found through other exceptions being thrown for class BOUNDED_STACK. One notices that, regardless of the timeout, the number of bugs found by contract violations is always higher than the number of bugs found through other exceptions. Furthermore, this latter number increases only slightly from timeout 1 to timeout 2, and then does not increase at all. For classes BANK_ACCOUNT and PRIMES, all version of the random generation algorithm constantly find more or an equal number of bugs through contract violations that through other exceptions.

Although these results are not directly relevant for estimating the performance of random testing, our intuition is that other testing

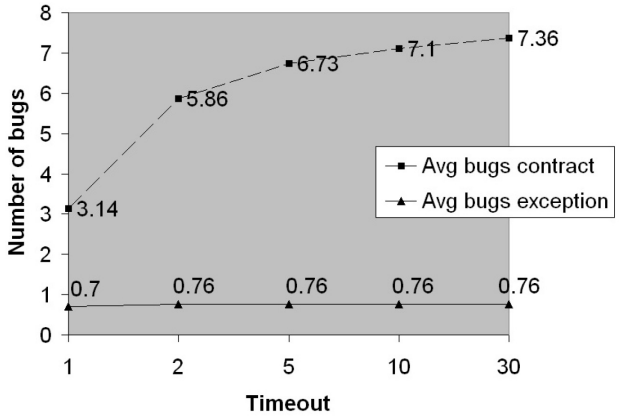

Figure 9: Evolution over time of the number of bugs found through contract violations and through other exceptions for class BOUNDED_STACK

strategies, especially guided ones, would achieve different distributions. For this reason we also provide these results in this paper, so that other studies (comparing *different* testing strategies) can benefit from them.

4. DISCUSSION

4.1 Methodology

We chose to use contracts and exceptions as automated oracles. One can argue that this strategy is neither complete (bugs which do not trigger contract violations or other exceptions are missed) nor sound (false positives might be reported). The former is a valid point, but the size of the study made it impossible to perform manual evaluation of all generated tests. The latter is a matter of convention: a contract violation always signals a bug; whether this bug is in the contract or in the implementation does not make a difference at this point. Bugs in the specification are just as dangerous as bugs in the implementation because they too prove the presence of an error in the developer's thinking process. They are also dangerous because a programmer trying to use the functionality of that software and looking at its interface (in which the contracts are included) will make wrong assumptions about the intended specification of that software.

One aspect in which our study and the ones mentioned in section 5 differ is the criterion used to evaluate the performance of the examined testing strategy. In our case, this is the number of bugs found in a fixed period of time. Other commonly used criteria are the time to first bug (time elapsed until the first bug is found), number of tests run until the first bug-revealing test case is generated, proportion of fault-revealing tests out of total tests generated, code and data coverage of the generated tests, number of false positives (false bug alarms), etc. Although all these measures are intuitively relevant, we consider number of found bugs to be more important than any of them, as this should be the purpose of any unit testing strategy: finding bugs (if there are any) in the software units under test.

4.2 Threats to validity

As is the case for any experimental study, the conclusiveness of the results depends on the representativeness of the samples examined. Testing a higher number of classes would naturally have in-

Table 5: Probability values that maximize the number of found bugs for each timeout

	Timeout=2		Timeout=5		Timeout=10		Timeout=30	
Class name	P_{New}	P_{Basic}	P_{New}	P_{Basic}	P_{New}	P_{Basic}	P_{New}	P_{Basic}
BANK_ACCOUNT	0.25	?	0.25	0; 0.25	0.25	0.25	0.25; 0.5	0; 0.25
BOUNDED_STACK	0.5; 0.75	0.5	0.75	0.75	0.75	0.75	0.75	0.75
HASH_TABLE	0.25; 0.5	>0	0.25;0.5	0.75	0.25; 0.5	0.5; 0.75	0.25; 0.5	0.5; 0.75
PRIMES	1	0.75; 1	1	0.75	1	0.75	1	1
STRING	0.25	0.75	0.25	0.25	0.25	0	0.25	0
UTILS	?	0	0.75	0	0.75	0	0.5; 0.75	0; 0.25

creased the reliability of the results. The very high number of tests that we had to run for each class in order to explore all possible combinations of parameters (the probabilities and the seed) and the duration of each such test (30 minutes) made it impossible for us to test more classes in the time during which we had exclusive access to the hardware necessary for the experiment. Hence, we chose the classes that we tested so that they come from different sources, implement different concepts and functionality, were produced by programmers with different levels of expertise, and have different sizes (in lines of code). Despite this, these classes do not exhibit all types of bugs that can be found in object-oriented software; hence, it is likely that, for some of these bugs, the behavior of a random testing algorithm would be slightly different.

As the results indicate, the seed can have a high influence on the number of found bugs. Each experiment was executed for three seeds. Using a higher number of seed values would have improved the robustness of the results, but practical time constraints prevented this. We plan to conduct a further study in which to use a higher number of seed values and investigate questions such as: is there a bound on the testing timeout so that, when testing for that duration, the choice of seed makes no difference? (in other words, the same bugs are found regardless of the used seed); how many seeds should be used over short timeouts to overcome the variability of the results?

Perhaps the greatest threat to the validity of our results is the existence of bugs in the testing tool that we used in the study. We are not aware of any such bugs, but a critical mind should not exclude the possibility of their presence.

4.3 Other flavors of random testing

The study presented in this paper is naturally not exhaustive. Other variations of the random generation algorithm are also possible. One can have different values for the probabilities or different algorithms. An example of a completely different algorithm which performs purely random testing is the one mentioned in section 2 which uses direct setting of object field values to get inputs for the tests.

Other possible variations go in the direction of making random testing "a little less random". An example would be to use, for basic types, constants taken verbatim from the source code, instead of the predefined, fixed set of values which we use in our algorithm. Another promising direction consists of using dynamic inference of abstract types [16]. This method finds sets of related variables (such as variables declared as integers which actually represent sums of money, or others declared as integers which represent ages of people). Determining these sets allows for them to be treated differently by the testing strategy.

This is a practical limitation which any experimental study has and it is part of the composable nature of such studies: the results of one case study can be used as the starting basis of another one, which adds to the scope of the first to make it more comprehensive.

5. RELATED WORK

A comprehensive overview of random testing is provided by Hamlet [18]. He discusses the advantages of purposely being "unsystematic" in choosing test inputs and also some of the theoretical deficiencies of the method. One of the interesting conclusions of the article is that the common misconceptions behind the criticism of random testing are most often triggered by misapplications of the method.

Although random testing of numerical applications has a longer standing tradition than random testing of object-oriented software, the interest for the latter has increased in recent years. Tools like JCrasher [12], Eclat [23], Jtest [2], Jartege [22], or RUTE-J [4] are proof of this interest.

Several tools combine random testing with other strategies. DART [15] combines random testing and symbolic execution. Agitar Software's Agitator [5] combines several strategies: static analysis, random input generation, and heuristics to find data likely to expose bugs.

There exist also directions of research based on the idea of random testing, but which try to improve its performance by adding some guidance to the algorithm. This is the case for Adaptive Random Testing [7] and its recent extension to object-oriented software [11], and for quasi-random testing [8].

Some studies of random testing are available, but unfortunately the vast majority are not recent. Some such studies ([14], [17], [24], [9]) compare random testing and partition testing. Their results are centered around the conditions under which partition testing (with its several flavors such as data-flow-oriented testing, path testing, etc.) can perform better than random testing. Their empirical investigations (or, in the case of Hamlet and Taylor [17], theoretical studies) also show that, outside of these restraining conditions, random testing outperforms partition testing.

Mankefors et al. [20] also investigate random testing and introduce a new method for estimating the quality of random tests. As opposed to the study presented here, their focus is on random testing of numerical routines and on quality estimations for tests which do not reveal bugs.

In a recently published report [3], Andrews et al. state that the main reasons behind the so far poor adoption of random generation for object-oriented unit tests is the lack of tools and of a set of recognized best practices. The authors provide such a set of best practices and also compare the performance of random testing to that of model checking. They show that random testing produces good results both when used on its own and when used as preparation for model checking.

D'Amorim et al [13] compare the performance of two input generation strategies (random generation and symbolic execution) combined with two strategies for test result classification (the use of operational models and of uncaught exceptions). For the results presented in this paper, one aspect of their study is particularly rel-

evant: the comparison of the test generation techniques (the random one and symbolic execution). The results of the study show much lower applicability of the symbolic-execution-based strategy than of the random one: 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. We see this as a very serious limitation of the strategy based on symbolic execution. Although, as the study shows, the symbolic-execution-based strategy does find bugs that the random one does not, the tool has extremely restricted practical applicability.

6. CONCLUSIONS

We have presented the results of an extensive case study that evaluates the performance of random unit testing of object-oriented applications over fixed timeouts. The purpose of the study was to see how random unit testing performs in general and to determine a most effective strategy (out of the examined ones) to recommend as best practice.

The evolution of the number of found bugs is inversely proportional to the elapsed time. Especially surprising is the steepness of the increase in the number of found bugs over the first few minutes of testing.

Because we used contracts and thrown exceptions as an automated oracle, we also examined the question of what proportion of bugs are found due to each of these factors. Our results indicate that over longer timeouts (10 minutes and more) the number of bugs found through contract violations is much higher than that of bugs found through other exceptions. For small timeouts (1 or 2 minutes) the situation is reversed.

In particular, the study evaluated implementation choices on the input generation algorithm: the frequency with which we create new objects as opposed to using existing ones, the frequency with which we try to diversify the object pool, and whether basic values are generated completely randomly or selected from a fixed predefined set. The combination of factors that gives the best overall result is 0.25 for creating new objects (so a creation of a new object once every 4 test cases are run), 0.5 for the probability of diversifying (the frequency with which we perform diversification operations on the objects in the pool), and 0.25 for the probability of generating values for basic types randomly as opposed to selecting these values from a fixed, predefined set. The process can be fine tuned depending on the classes that are tested. This fine-tuning needed by a class is not dependent on the timeout value chosen. In future experiments it would be interesting to check if the result is also valid on classes that evolve, i.e. it is applicable to future versions of the same class and, if so, under what circumstances.

The results of the study also show that the values chosen for the above-mentioned parameters can have a significant impact on the performance of the testing tool. Although there exists a combination of parameter values that seems to deliver overall good results, it is recommendable to run the tool with several combinations of parameter values, since different such combinations may find different bugs.

The study presented in this paper answered several questions about the performance of random testing in general and about the factors that influence it. It has shown that, despite its simplicity and unguided nature, random testing does indeed find bugs, not only seeded ones, but also bugs present in widely used, industrial-grade code. Of particular importance is the observation that random testing finds a very high number of bugs in the *first few minutes* of testing a certain class. This indicates that, although this strategy might not find *all* bugs present in the code, its *rate* of finding bugs over short timeouts makes it a very good candidate for combining with other testing strategies, more expensive in terms of the computational resources they require.

The question of how random testing compares to other testing strategies is still open. We consider that providing an answer to this question is of utmost importance, because the great variety of automated testing strategies that are now available leaves developers and testers wondering as to the choice of testing tool that would deliver the best results for their projects. Future work includes extensive experimental studies to answer this question and the development of methods and metrics to compare testing strategies.

Acknowledgements

We thank Gustavo Alonso for providing us the hardware infrastructure that we used in the experiment.

7. REFERENCES

[1] The EiffelBase library. Eiffel Software Inc. http://www.eiffel.com/.

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

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

[4] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li. Tool support for randomized unit testing. In *RT '06: Proceedings of the 1st International Workshop on Random Testing*, pages 36–45, New York, NY, USA, 2006. ACM Press.

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

[6] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rome, Italy*, 2002.

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

[8] T. Y. Chen and R. Merkel. Quasi-random testing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 309–312, New York, NY, USA, 2005. ACM Press.

[9] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

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

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *RT '06: Proceedings of the 1st International Workshop on Random Testing*, pages 55–63, New York, NY, USA, 2006. ACM Press.

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

[13] M. d'Amorim, C. Pacheco, D. Marinov, T. Xie, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *ASE 2006: Proceedings of the 21st Annual International Conference on Automated Software Engineering*, Tokyo, Japan, September 20–22, 2006.

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

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[16] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, New York, NY, USA, 2006. ACM Press.

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

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

[19] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: the AutoTest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, January 3-6, 2007.

[20] S. Mankefors, R. Torkar, and A. Boklund. New quality estimations in random testing. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 468-478, 2003.

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

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

[23] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 25–29, 2005.

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