A

A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite

Gordon Fraser, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
Gordon.Fraser@sheffield.ac.uk
Andrea Arcuri, Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
arcuri@simula.no

Research on software testing produces many innovative automated techniques, but because software testing is by necessity incomplete and approximate, any new technique faces the challenge of an empirical assessment. In the past, we have demonstrated scientific advance in automated unit test generation with the EVOSUITE tool by evaluating it on manually selected open source projects or examples that represent a particular problem addressed by the underlying technique. However, demonstrating scientific advance is not necessarily the same as demonstrating practical value: Even if EVOSUITE worked well on the software projects we selected for evaluation, it might not scale up to the complexity of real systems. Ideally, one would use large "real-world" software systems to minimize the threats to external validity when evaluating research tools. However, neither choosing such software systems nor applying research prototypes to them are trivial tasks.

In this paper we present the results of a large experiment in unit test generation using the EVOSUITE tool on 100 randomly chosen open source projects, the 10 most popular open source projects according to the SourceForge website, 7 industrial projects, and 11 automatically generated software projects. The study confirms that EVOSUITE can achieve good levels of branch coverage (on average 71% per class) in practice. However, the study also exemplifies how the choice of software systems for an empirical study can influence the results of the experiments, which can serve to inform researchers to make more conscious choices in the selection of software system subjects. Furthermore, our experiments demonstrate how practical limitations interfere with scientific advances: Branch coverage on an unbiased sample is affected by predominant environmental dependencies. The surprisingly large effect of such practical engineering problems in unit testing will hopefully lead to a larger appreciation of work in this area, thus supporting transfer of knowledge from software testing research to practice.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Experimentation, Reliability

Additional Key Words and Phrases: Unit testing, automated test generation, branch coverage, empirical software engineering, JUnit, Java, benchmark

1. INTRODUCTION

Software testing is an essential yet expensive activity in software development, therefore much research effort has been put into automating as many aspects as possible, including test generation. For "simple" test generation techniques such as random testing, it is possible to provide rigorous answers based on theoretical analyses (e.g.,

This work is funded by a Google Focused Research Award on "Test Amplification", the EPSRC project "EXOGEN" (EP/K030353/1), and the Norwegian Research Council.

DOI:http://dx.doi.org/10.1145/0000000.0000000

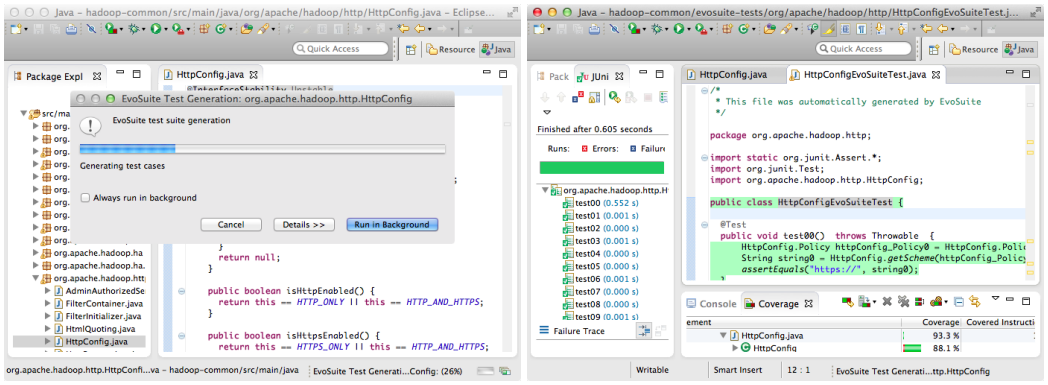
see [Arcuri et al. 2012]). For more complex techniques where mathematical proofs become infeasible or too hard, researchers have to rely on empirical analyses. There are several challenges when carrying out empirical studies. To demonstrate that a new technique improves over the state of the art in research often a handful of examples representing the targeted problem domain are sufficient. Yet, if a technique works well in the lab on a specific software system, will it also work well when it is applied on other software developed by practitioners?

In the past, we have demonstrated that the EVOSUITE unit test generation tool [Fraser and Arcuri 2011b; Fraser and Arcuri 2013c] can automatically produce test suites with high code coverage for Java classes. As EVOSUITE only requires compilable source code, we were able to select software from some of the many open source projects in freely accessible repositories (e.g., SourceForge[1] or Google Code[2]). In principle, the possibility to use abundantly available open source software can serve to strengthen an evaluation, as evaluating a test generation technique for a specific formalism would restrict the choice to examples of that formalism (which often equates to a choice from what has been used in the literature of that domain previously). Similarly, obtaining real data from industry is a very difficult and time consuming activity. However, even when using open source projects for experimentation, there remain unavoidable threats to the external validity of these experiments: How well do our results generalize to other programs than the ones chosen by us? Was our selection large enough? Even when the selection of employed software systems is *large* and *variegated*, certain classes of problems might not be present. For example, if a proposed testing technique does not support file system I/O, then that kind of software might have been excluded from the empirical study. Thus, was our selection unconsciously biased? How well does EVOSUITE actually work?

To shed light on these questions, in this paper we present an empirical study where we aimed to minimize the threats to external validity by making the choice of the employed artifacts statistically as sound as possible. We randomly selected 100 Java projects from SourceForge, a well established open source repository. As SourceForge is home to many old and stale software projects, we further considered the 10 most popular Java applications, which have been recently downloaded and used by millions of users. The resulting SF110 corpus of classes is not only large (110 projects, 23,886 classes, more than 800 thousand bytecode level branches and 6.6 millions of lines of code): The main virtue of this corpus is that, because it is randomly selected from an open source repository, the proportions of kinds of software (e.g., numerical applications and video games) are *statistically* representative for open source software. To provide further insight on how the choice of software artifacts influences experimental results and the conclusions one can draw from them, we also performed experiments on seven industrial systems (totaling 4,208 classes) and 11 automatically generated software projects [Park et al. 2012] (totaling 1,392 classes) in addition to SF110. Indeed, providing new empirical results on different types of software contributes to improve generalizability of our past findings on EVOSUITE, and it is common practice in many domains (e.g., [Koru et al. 2010; Koru et al. 2007]).

The results of our empirical analysis confirm that, as demonstrated by previous empirical studies, EVOSUITE can indeed achieve high branch coverage — but only on certain types of classes. In practice, dependencies on the environment (e.g., files, network, databases) seem to inhibit high branch coverage, a point in case that experimental results can diverge depending on whether the aim is to show scientific advance or practical relevance. In fact, even more so than potentially leading to threats to

[1] http://sourceforge.net/, accessed June 2014.
[2] http://code.google.com/, accessed June 2014.



(a) Generating tests (b) Resulting tests

Fig. 1: Test generation in Eclipse works on the click of a button, and generated tests are compatible with third party tools such as coverage analysers.

external validity of experiments, environmental interactions may be a *practical* threat to the execution environment when applying test generation tools to unknown software: As we learned through directories cluttered with randomly named files after experiments, randomly disappearing project files, and an unfortunate episode where one of the authors lost the entire contents of his home directory, applying test generation tools to unknown software can lead to interactions with the environment in unpredictable ways. To ensure that this does not happen (again), EVOSUITE uses a *sandbox* where potentially unsafe operations (e.g., class methods that take as input the name of a file to delete) are caught and properly taken care of. This allows us to measure the effect of such environmental interactions on the aim of the test generation tools. To show that environmental interactions are not a problem that applies only to EVOSUITE, we validated our findings using Randoop [Pacheco et al. 2007], a popular random test generation tool for Java.

Besides the insights on practically relevant problems, we can use our experimental results to investigate and discuss the effects of the choice of software artifacts, and quantify how size and bias can skew results. Although there is no golden rule on how to design the perfect empirical study, we hope our experiments will inspire researchers to more conscious choices in empirical study design in unit testing research and beyond, and will help researchers put experimental results into context. To this purpose, we provide our selection of 110 SourceForge projects as a benchmark to the community.

This paper is organized as follows: We start with a brief description of EVOSUITE and its treatment of environmental dependencies in terms of a custom security manager in Section 2. Section 3 describes the experimental setup and the results of these experiments in detail. Finally, Section 4 reflects on the insights on our attempts to reduce the threats to external validity, and implications for the choice of software artifacts for experiments.

2. THE EVOSUITE UNIT TEST GENERATION TOOL

EVOSUITE [Fraser and Arcuri 2011b; Fraser and Arcuri 2013c] automatically generates test suites for Java classes, targeting branch coverage and other coverage criteria (e.g., mutation testing [Fraser and Arcuri 2014]). EVOSUITE works at the Java bytecode level, i.e., it does not require source code. It is fully automated and requires no manually written test drivers or parameterized unit tests. For example, when EVOSUITE is used

from its Eclipse plugin (see Figure 1), a user just needs to select a class, and tests are generated with a mouse-click.

EvoSuite uses an evolutionary approach to derive these test suites: A genetic algorithm evolves candidate individuals (chromosomes) using operators inspired by natural evolution (e.g., selection, crossover and mutation), such that iteratively better solutions with respect to the optimization target (e.g., branch coverage) are produced. For details on this test generation approach we refer to [Fraser and Arcuri 2013c]. To improve performance further, we are investigating several extensions to EvoSuite. For example, EvoSuite can employ dynamic symbolic execution to handle the cases in which evolutionary techniques may struggle [Galeotti et al. 2013]. Recent comparisons as part of the unit testing tool competition [Bauersfeld et al. 2013; Fraser and Arcuri 2013a] in which EVOSUITE came first suggest that EvoSuite can be considered on par with the state of the art in testing Java classes.

One particular advantage of EvoSuite in the scope of the experiments described in this paper is that EvoSuite only requires the bytecode of the class under test (CUT), and the `jar` files of all the libraries it depends on should also be on the classpath. Running large experiments can be done conveniently through EvoSuite's command line interface, and EvoSuite automatically determines dependencies, and generates a JUnit test suite for each CUT. This level of automation was important for our experiments, as it allowed us to apply the tool to new software projects out of the box, without any necessary manual adaptation.

As the generated unit tests are meant for human consumption, EvoSuite applies various post-processing steps to improve readability (e.g., minimising) and adds test assertions that capture the current behavior of the tested classes. To select the most effective assertions, EvoSuite uses mutation analysis [Fraser and Zeller 2012]. For more details on the tool and its abilities we refer to [Fraser and Arcuri 2011b], and for more implementation details we refer to [Fraser and Arcuri 2013b].

2.1. The Java Security Manager

Most unit test generation techniques require executing code. If this code interacts with its environment, then not only may achieving high branch coverage be difficult, but also unexpected or undesirable side-effects might occur. For example, the code might access the filesystem or network, causing damage to data or affecting other users on the network. To overcome this problem, EvoSuite provides its own custom *security manager*: The Java language is designed with a permission system, such that potentially undesired actions first ask a security manager for permission. EvoSuite uses its own security manager that can be activated to restrict test execution. It is also important to find out to what extent these unsafe operations are a problem for test generation. Consequently, we kept track of which kinds of permissions were requested from the code under test.

One of the original applications of the Java language was the development of "applets", i.e., programs included on web pages and run inside the browser. In this case, a security manager is of paramount importance, otherwise malicious websites could host applets that are developed on purpose for example to take control/damage the user's computer, steal confidential information, etc. The need of a security manager in automated test data generation is different from the case of Java applets. For example, when testing one's own code, a software engineer does not really need to be afraid of his code "stealing" information. Furthermore, crashing the testing tool (e.g., the CUT starting to "stop" all the threads in the running JVM) is of less concern if the alternative is to have a restrictive security manager that would prevent generating any test data for the given CUT. At the end of the day, what really matters are the side-effects outside the JVM, such as deleting files, sending messages on a network, etc. Note that besides safety

considerations, allowing tests to affect the environment would also create possible dependencies between tests, and may cause generated tests to fail when executed outside the test generation environment.

In our previous experiments in the conference version of this paper [Fraser and Arcuri 2012], no permissions were granted, except for three permissions which we determined necessary to run most code in the first place in our earlier experiments [Fraser and Arcuri 2013c]: (1) Reading from properties, (2) Loading classes, and (3) Reflection. Except for these permissions, all other permissions were denied. As discussed above, this might be overly strict. For the experiments in this paper, we first analyzed *all* possible permissions that can be requested during the execution of the CUT. For each permission, we decided whether (1) it was safe enough to grant it, (2) we should deny it straight away (a clear example is deleting files), or (3) grant it but apply some techniques to "re-set" the environment after a test case is executed.

All permissions in Java extend the class `java.security.Permission`, which has at least 25 derived concrete subclasses. Each permission can have a *name* and an *action*. For example, for the permission class `FilePermission`, actions are `read`, `write`, `execute` and `delete`, whereas the name represents the path of the file the permission is asked for. On the other hand, a permission such as `RuntimePermission` has no action, but at least 27 different "names" representing permissions such as accessing environment variables (i.e., `getenv`) and modifying running threads (e.g., `modifyThread` and `modifyThreadGroup`).

Beside the permissions in the Java API, the CUT itself (and its third-party libraries) could define and use its own custom permissions. This is achieved by extending the class `java.security.Permission`, and then calling the method `checkPermission(Permission perm)` on the security manager whenever such permissions need to be checked. Custom permissions are not very common, but they did indeed appear in the software systems used in this paper.

Table I shows all the permissions we decided to grant in EvoSuite. Choosing which permissions to grant and which to deny requires informed judgment, which forced us to study all those permissions in detail. There is a tradeoff between security and test effectiveness, and granting a permission always incurs some risks. As a rule of thumb, we tended to allow "reading" operations, but checked in detail the ones that would result in modifying the state (both of the JVM and its environment).

Among the permissions granted in Table I, the action `write` in `PropertyPermission` is treated slightly specially: We allowed the CUTs to modify all properties except for a fixed set of system properties (e.g., `java.vm`) defined in the Java API documentation[3].

Here, we provide explanations on why we decided to deny some of the Java permissions during test case generation. We do not claim that these decisions are optimal, and it might well be that in some cases we have been overprotective, denying permissions that might not be harmful after all. In some cases, we denied permissions just because we did not fully understand their implications and they were so rare that it did not warrant more investigation at that time.

If our security manager is called on a security permission that the manager is not aware of (e.g., we did not know about the existence of that permission, or EvoSuite is called on a new version of Java that we do not support yet), that permission will be denied, but only if it is in the `java*` package. The reason is that, on one hand, we do want to deny permissions for which we could not verify whether they are harmful or not. On the other hand, we need to allow user's defined permissions, as those are harmless. Checking for the `java*` package is a possible way to distinguish among those two cases.

In the following, we discuss all permissions in detail:

[3]http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#getProperties(), accessed March 2014

Table I: List of security permissions that we granted for the experiments reported in this paper.

Class	Allowed Operations/Actions
CUT permissions	All
java.lang.RuntimePermission	accessClassInPackage*, accessDeclaredMembers, charsetProvider, createClassLoader, defineClassInPackage*, getClassLoader, getenv.*, getFileSystemAttributes, getProtectionDomain, getStackTrace, loadLibrary.awt, loadLibrary.cmm, loadLibrary.fontmanager, loadLibrary.instrument, loadLibrary.laf, loadLibrary.lcms, loadLibrary.j2pkcs11, loadLibrary.jawt, loadLibrary.jaybird, loadLibrary.jpeg, loadLibrary.kcms, loadLibrary.management, loadLibrary.net, loadLibrary.nio, loadLibrary.sunec, loadLibrary.t2k, loadLibrary.*libmawt.so, loadLibrary.*liblwawt.dylib, modifyThread, modifyThreadGroup, readFileDescriptor, reflectionFactoryAccess, selectorProvider, setContextClassLoader, setDefaultUncaughtExceptionHandler, setFactory, setIO, stopThread
java.security.SecurityPermission	getDomainCombiner, getPolicy, printIdentity, getProperty.*, putProviderProperty.*, getSignerPrivateKey
java.awt.AWTPermission	All, but only if property java.awt.headless is defined
java.lang.management.ManagementPermission	monitor
javax.net.ssl.SSLPermission	getSSLSessionContext
java.io.FilePermission	read
java.io.SerializablePermission	All
java.lang.reflect.ReflectPermission	All
java.net.NetPermission	All
javax.security.auth.kerberos.ServicePermission	All
javax.security.auth.PrivateCredentialPermission	All
javax.sound.sampled.AudioPermission	All
java.util.logging.LoggingPermission	All
java.util.PropertyPermission	All
javax.management.MBeanPermission	All
javax.management.MBeanServerPermission	All
javax.management.MBeanTrustPermission	All
javax.management.remote.SubjectDelegationPermission	All

— `java.security.AllPermission` is obviously denied, as it implies all the other permissions.

— All `java.net.SocketPermissions` are strictly denied. Our industrial partners were particularly worried about this kind of permission, as having unpredictable TCP/UDP activity on a corporate network was simply out of question. Furthermore, there are potential hazards if the tested software interacts with physical hardware over the network (e.g., a GUI application used to monitor a set of physical sensors/actuators), and if the developing machine of the engineers using EVOSUITE is on the same network (as it was the case with our industrial collaborators). Similarly, we denied `javax.xml.ws.WebServicePermission` and writing operations (`setHostnameVerifier` and `setDefaultSSLContext`) in `javax.net.ssl.SSLPermission`.

— `javax.security.auth.AuthPermission`, `javax.security.auth.kerberos.DelegationPermission` and `java.security.UnresolvedPermission` are cases of rare permissions we did not fully analyze, and so we deny them by default.

— For `java.lang.management.ManagementPermission`, we denied the action `control`, as it modifies properties of the running JVM.

— `java.sql.SQLPermission` writing actions are denied. Reading permissions could be granted in theory, but because EVOSUITE works at the unit level, relying on external processes like a database for unit tests would be unreliable, so we decided to deny that whole class of permissions.

— Writing actions in `java.security.SecurityPermission` were denied. This includes: `createAccessControlContext`, `setPolicy`, `createPolicy.*`, `setProperty.*`, `insertProvider.*`, `removeProvider.*`, `setSystemScope`, `setIdentityPublicKey`, `setIdentityInfo`, `addIdentityCertificate`, `removeIdentityCertificate`, `clearProviderProperties.*`, `removeProviderProperty.*` and `setSignerKeyPair`. However, we had

to grant `putProviderProperty.*`, as it is needed for some common Java API functionalities, albeit we do not know for sure if it has negative side-effects.

— `java.awt.AWTPermissions` were denied, unless EVOSUITE is run in headless mode (which is its default configuration). In headless mode, many GUI components needing the screen or mouse/keyboard interactions will just throw an exception, so there should not be any particular risk.

— `java.io.FilePermission` is among the most dangerous permissions to grant. All its actions but `read` were denied (in particular after we experienced loss of parts of our files). Technically, though, even a `read` action could be harmful. On operating systems like Windows where file reading operations put locks on those files, there can be potential for interferences with other processes running on the same machine that try to manipulate those same files. However, this problem does not occur on operating systems like OS X and Linux. There is one exception: we allowed the `write` action on the folder "$USER_DIR/.java/fonts". This is needed as many GUI classes do have a write operation on such folder during their loading (i.e., in their static initializer), which in normal conditions is always executed because inside a `AccessController.doPrivileged` block.

— `java.lang.RuntimePermission` has several types of actions. `setSecurityManager` and `createSecurityManager` were denied because they can modify the installed security manager, e.g., by disabling it. `exitVM` was denied as it would kill the JVM, and so EVOSUITE would just crash without generating any unit tests. `shutdownHooks` was denied, as threads executed at the exit of the JVM would not be part of what unit tests can handle. In principle, there could also be issues when thousands of generated test cases during the search register new shutdown hook threads, as all theses threads would be executed in parallel before the JVM quits. Because there is a limit to the number of threads that can be created on the same machine, there could be interferences with other processes running on the same machine and that try to spawn new threads as well. `writeFileDescriptor` was denied, as it is related to the manipulation of files and network sockets. `queuePrintJob` could lead to printing documents if a printer is connected, which is clearly not a desirable effect for a unit test generation tool. `preferences` can lead to changes on the file disk, and so was denied. `loadLibrary.*` was denied but for some known libraries shipped with the JVM and used in some specific classes. This action is particularly tricky, as it is loads native code (e.g., a compiled C library). Native code would be executed outside of the security manager, and so it is highly risky (e.g., it can delete files).

3. A LARGE EMPIRICAL STUDY WITH EVOSUITE

The performance of test generation tools is commonly evaluated in terms of the achieved code coverage (e.g., statement or branch coverage). High code coverage by itself is not sufficient in order to find defects as there are further major obstacles, most prominently the oracle problem: Except for special kinds of defects, such as program crashes or undeclared exceptions, the tester has to provide an oracle that decides whether a given test run detected an error or not. This oracle could be anything from a formal specification, test assertions, up to manual assessment. The oracle problem entails further problems; for example, in order to be able to come up with a test assertion a generated test case needs to be understandable and preferably short. However, in all cases a prerequisite to the oracle problem is to find an input that takes the program to a desired state. Therefore, in our experiments we compare the results in terms of the achieved code coverage. In particular, we focus on branch coverage. EVOSUITE measures branch coverage at the bytecode level, where complex branching conditions are compiled to several atomic conditions; each of these needs to evaluate to true and to false (see [Li et al. 2013] for a detailed discussion of bytecode level branch coverage). In

Table II: Top 10 most popular projects on SourceForge, October 2012. The table reports how many million times a project was downloaded, and since when the project has been hosted.

Rank	Name	Downloads	Creation Date	Brief Description
1	netweaver	11.3M	2010-04-29	Eclipse server adapter for the SAP NetWeaver Application Server Java.
2	squirrel-sql	5.1M	2001-05-31	Graphical SQL client to view the structure of a JDBC compliant database.
3	sweethome3d	30.9M	2005-11-07	Interior design application to draw the 2D plan of a house and view the results in 3D.
4	vuze	535.1M	2003-06-24	P2P file sharing client using the bittorrent protocol.
5	freemind	15.9M	2000-06-18	A mind mapper and hierarchical editor with strong emphasis on folding.
6	checkstyle	11.2M	2003-05-03	Eclipse plug-in that provides feedback to the user about violations of coding style rules.
7	weka	3.4M	2000-04-27	Collection of machine learning algorithms for solving data mining problems.
8	liferay	6.8M	2002-03-18	Enterprise portal framework, offering Web publishing, content management, etc.
9	pdfsam	5.3M	2006-02-15	Tool to merge and split pdf documents.
10	firebird	12.0M	2000-07-30	A multi-platform relational database management system supporting ANSI SQL.

order to consider all edges in the control flow graph as coverable items, any methods with no branching statements also need to be covered. We do not currently consider exceptional control flow edges.

Furthermore, as discussed, executing unknown code can be unsafe, and environmental interactions may interfere with the traditional goal of code coverage. To this extent, we analyze environmental dependencies and interactions during these experiments. Based on these measurements, we aim to answer the following research questions:

RQ1: What is the probability distribution of achievable branch coverage with EvoSuite on open source software?

RQ2: Is the branch coverage achieved by EvoSuite affected by environmental dependencies?

RQ3: Do the findings based on EvoSuite generalize to other testing tools?

RQ4: What are the differences between a random choice and choosing the most popular software for an empirical study using EvoSuite?

RQ5: How do unit testing results for EvoSuite differ between SF110 and industrial software?

RQ6: How do unit testing results for EvoSuite differ between SF110 and automatically generated software?

RQ7: How does the class and project size affect the difficulty of unit testing a class with EvoSuite?

3.1. Artifact Selection

3.1.1. Open Source Software. To select an unbiased sample of Java software, we consider the SourceForge open source development platform. SourceForge provides infrastructure for open source developers, ranging from source code repositories, webspace, discussion forums, to bug tracking systems. There are other similar services on the web, for example Google Code, GitHub, or Assembla. We chose SourceForge because it has a long history and for a long time it was the dominant site of this type.

We based our selection on a dataset of all projects tagged as being written in the Java programming language, collected using a web crawler. In total there were 48,109 such projects at the time of our experiment, and applying EVOSUITE to all of them would not be possible in reasonable time. Therefore, we sampled the dataset, picking one randomly chosen project out of this data set at a time. For each chosen project we downloaded the most recent sources from the corresponding source repository and tried to build the program. It turned out that many projects on SourceForge have no files (i.e., they were created but then no files were ever added). A small number of projects was also misclassified by their developers as Java projects although in fact they were written in a different programming language. Finally, sometimes they were too old and relying on particular Java APIs that are no longer available. In total, we therefore had to randomly sample 321 projects until we had a set of 100 projects in binary format.[4] This selection leads to an experiment that is sound in the narrow context of SourceForge-hosted open source software, because the projects that are subject of the experiment are chosen in an *unbiased* way: a well defined methodology was employed to select those subjects, rather than manually selecting them.

One might argue that projects randomly sampled from SourceForge may include projects that are never used, poorly designed and implemented, and projects developed by students or non-professionals. Although we would counter that non-professional developers might particularly benefit from automated test generation, we take this potential criticism into account and also consider the 10 *most popular* Java programs on SourceForge as follows: When searching for programs on SourceForge, there is also the option to sort them by recent popularity[5]. Our sampling was done on the 12th of October 2012 and, as such, at the time of reading this paper that popularity ranking might have changed. Table II shows details of these 10 projects. Not surprisingly, these projects have been downloaded millions of times, by people from all around the world. Choosing the 10 most popular Java programs was a biased selection, as we deliberately aimed at including in our empirical study *programs that matter* for the final users, regardless of their size and complexity. This is a way to address the possible limitations of only sampling programs at random.

We call this combination of 100 plus 10 projects the SF110 corpus of classes. Table III shows several statistics per each of these 110 projects (e.g., the number of classes and branches). Table IV summarizes those statistics on all 110 projects. When we only refer to the 100 projects selected at random, we use the term *Rand100*, whereas the top most popular projects are labeled *Top10*.

Table V presents the summarized statistics (e.g, mean and standard deviation) for the entire SF110 corpus, as well as for the subsets *Rand100* and *Top10*. These numbers were derived using EVOSUITE, which only lists top-level classes; EVOSUITE attempts to cover member or anonymous classes together with their parent classes. Furthermore, EVOSUITE might exclude certain classes it determines that it cannot handle, such as non-public classes. In total, there are 23,886 classes and more than 800 thousand bytecode branches reported by EVOSUITE in the SF110 corpus. Both in terms of the number of classes and branches, what stands out is the large variation in the data; e.g., the number of classes in a project ranges from 1 to 6,984, and the number of branches in a class ranges from 0 to 12,879. Furthermore, these distributions present infrequent extreme deviations, which is represented by high kurtosis values, and are

[4]The details of this selection process and the code of those projects are available online at http://www.evosuite.org/SF110

[5]Note that there could be other ways to choose these 10 projects. For example, set a threshold for minimum number of downloads, or minimum number of ratings, and select randomly from all the projects that achieved the threshold. As we had to make a choice, we took the one that looked the least biased in our opinion.

Table III: Details of the SF110 corpus. For each project, we report how many testable classes it is composed of, the total number of bytecode branches, the number of `.java` source code files and lines of code (LOC) in them. We also report the number of non-commenting source statements (NCSS) and the average cyclomatic complexity number (McCabe metric) per method, using the JavaNCSS tool (version 32.53). We also report the total number of third-party jar libraries each project uses.

Name	# Testable Classes	# Branches	# Java Files	LOCs	NCSS	Avg. CCN	# Jar Files
liferay	6,984	321,443	8,345	2,878,331	922,463	1.73	305
ifx-framework	3,900	26,486	4,027	378,315	71,626	1.01	4
vuze	2,331	108,961	3,304	822,100	322,411	2.67	7
squirrel-sql	934	23,407	1,151	213,498	76,815	2.00	49
weka	888	61,269	1,031	462,994	179,139	3.05	4
caloriecount	596	12,197	684	103,582	37,413	1.74	3
openjms	542	18,266	624	132,586	40,511	2.80	25
summa	512	16,543	584	119,963	47,866	3.03	106
freemind	422	13,298	468	100,237	45,120	2.15	46
jcvi-javacommon	396	7,434	619	89,198	30,472	1.97	0
noen	382	4,029	408	36,566	14,155	1.74	142
pdfsam	348	8,140	369	66,383	27,746	2.66	30
corina	334	7,733	349	78,144	33,034	2.54	32
lilith	270	7,288	295	60,064	28,689	2.35	66
firebird	210	13,212	258	76,886	30,277	2.76	15
at-robots2-j	197	2,046	231	13,278	6,877	1.41	1
jsecurity	197	4,182	298	34,673	9,470	2.05	38
netweaver	187	7,256	204	38,016	17,953	2.82	61
xbus	181	4,645	203	38,816	13,294	3.50	38
jiggler	175	6,699	184	40,462	16,403	2.85	2
heal	170	6,132	184	32,983	16,982	2.82	29
sweethome3d	164	13,640	185	96,728	47,233	2.63	19
checkstyle	154	2,336	169	36,795	12,548	2.44	30
dom4j	146	5,597	173	42,198	12,980	1.87	11
hft-bomberman	126	1,954	135	14,468	6,474	1.87	29
quickserver	124	4,953	152	26,675	12,880	2.66	8
jdbacl	112	5,668	126	28,618	13,296	2.50	25
gangup	103	2,527	95	20,629	8,498	2.22	8
jiprof	103	5,683	113	26,296	10,473	3.76	5
lhamacaw	99	2,937	108	36,300	14,801	2.23	1
wheelwebtool	99	6,968	113	29,761	12,787	3.40	8
db-everywhere	97	2,099	104	11,079	5,686	2.54	36
twfbplayer	86	1,572	104	14,642	5,559	1.72	2
echodep	82	3,323	81	26,708	10,381	4.87	37
openhre	82	2,471	135	17,676	6,550	1.88	33
lagoon	79	2,512	81	17,415	6,060	3.52	18
objectexplorer	78	1,557	88	12,534	5,199	1.69	3
beanbin	76	996	88	4,784	2,878	2.13	34
schemaspy	71	3,407	72	16,157	7,987	3.09	0
fim1	69	2,042	70	13,713	8,273	2.21	1
jhandballmoves	69	1,455	73	8,553	4,005	2.32	2
nutzenportfolio	60	1,832	84	14,794	6,052	1.67	30
follow	58	633	60	7,634	3,003	1.79	0
jwbf	57	1,288	69	10,970	4,094	2.33	18
geo-google	56	892	62	20,974	3,941	1.18	11
lotus	54	226	54	1,354	681	1.94	0
petsoar	54	508	76	5,541	1,695	1.55	31
jnfe	51	289	68	5,199	1,294	1.38	60
javathena	50	2,228	53	13,927	6,194	1.84	4
gfarcegestionfa	48	760	50	5,480	2,923	2.18	4
lavalamp	48	313	54	2,123	1,039	1.50	16
water-simulator	48	736	49	9,931	5,433	2.41	6
apbsmem	47	1,269	50	9,342	3,831	2.92	3
a4j	45	954	45	6,618	2,787	1.80	2
javabullboard	44	2,295	44	13,987	5,647	2.85	34

Table III: Details of the SF110 corpus (continued).

Name	# Testable Classes	# Branches	# Java Files	LOCs	NCSS	Avg. CCN	# Jar Files
jtailgui	43	433	44	4,053	1,409	2.01	12
ext4j	42	579	45	3,371	1,508	2.20	23
newzgrabber	39	1,367	39	6,603	4,007	3.95	0
jmca	38	10,487	25	16,891	10,610	7.57	0
xisemele	38	339	56	5,766	1,399	1.29	0
jopenchart	37	774	44	7,146	2,397	1.79	2
feudalismgame	35	1,352	36	4,355	2,653	3.34	0
mygrid	34	1,006	37	4,539	2,360	2.82	8
shop	34	1,171	34	5,348	2,741	3.80	0
bpmail	32	370	37	2,765	1,252	1.58	39
glengineer	32	996	41	5,694	2,026	2.20	0
dsachat	31	943	32	5,546	2,993	3.24	4
jaw-br	30	631	30	6,006	3,342	1.59	20
sugar	29	771	37	5,516	2,180	2.99	6
inspirento	27	510	36	5,290	1,769	1.76	2
jni-inchi	24	357	24	3,100	783	2.05	3
fixsuite	22	480	25	4,897	2,088	1.99	4
biblestudy	21	579	21	3,178	1,683	1.96	2
asphodel	19	141	24	1,358	520	1.57	6
httpanalyzer	19	327	19	4,928	2,472	2.03	9
jgaap	19	177	17	1,451	815	1.84	0
sfmis	19	268	19	1,749	941	1.56	49
templateit	19	595	19	3,315	1,542	2.82	6
tullibee	18	1,191	20	4,388	2,449	3.78	0
imsmart	17	155	20	1,407	790	1.82	65
diebierse	16	350	20	2,482	1,539	1.74	3
dash-framework	15	39	22	776	166	1.43	5
gsftp	15	553	17	3,441	1,785	2.85	2
io-project	15	160	19	2,136	485	1.70	4
javaviewcontrol	15	3,111	17	5,953	3,844	7.19	7
omjstate	14	103	23	1,628	387	1.51	12
rif	14	167	15	1,902	693	3.01	37
byuic	12	2,958	12	7,699	4,909	10.29	2
fps370	12	312	8	2,400	1,056	2.44	4
battlecry	11	663	11	3,342	2,208	3.89	0
celwars2009	11	684	11	3,654	2,052	4.09	0
saxpath	11	1,079	16	4,578	1,441	2.10	4
diffi	10	150	10	851	392	2.35	0
gaj	10	66	14	404	187	1.21	0
ipcalculator	10	448	10	4,201	2,132	2.37	0
dvd-homevideo	9	370	9	4,204	2,289	3.19	0
falselight	8	25	8	732	297	2.16	1
gae-app-manager	8	90	8	646	257	2.56	9
nekomud	8	65	10	695	270	2.13	8
resources4j	7	222	14	2,270	938	2.34	0
biff	6	831	3	2,371	1,753	12.36	0
classviewer	6	418	7	2,966	1,258	3.83	0
dcparseargs	6	100	6	387	158	3.57	0
sbmlreader2	6	87	6	1,025	400	3.43	5
trans-locator	5	66	5	454	298	1.57	0
jclo	4	133	3	602	276	2.88	0
shp2kml	4	39	4	334	196	1.72	29
jipa	3	163	3	619	304	4.59	0
greencow	1	1	1	16	4	1.00	0
templatedetails	1	18	1	513	262	1.29	15

Table IV: Summarized statistics of the whole SF110 corpus.

Property	Value
Number of Projects:	110
Number of Testable Classes:	23,886
Number of Target Branches:	808,056
Number of Java Files:	27,997
Lines of Code:	6,628,619
Non-Commenting Source Statements:	2,340,843
Average Cyclomatic Complexity Number:	2.63
Number of Jar File Libraries:	1,939

Table V: For the SF110 corpus, statistical details of the number of classes per project and number of bytecode branches per class. For median, average, skewness and kurtosis, we also report a 95% confidence interval (CI) calculated with a 10,000 run bootstrapping.

	# of Classes per Project			# of Branches per Class		
Statistics	Rand100	Top10	All	Rand100	Top10	All
Min	1	154	1	0	0	0
Median	37.5	385	43.5	7	14	10
Median CI	[27.00, 48.50]	[-839.50, 583.00]	[32.00, 55.00]	[7.00, 7.00]	[13.00, 15.00]	[9.00, 10.00]
Mean	112.64	1262.20	217.15	20.87	45.39	33.83
Mean CI	[24.04, 167.00]	[-102.80, 2183.89]	[50.23, 336.34]	[18.40, 22.79]	[42.33, 47.91]	[31.84, 35.51]
Std. Deviation	398.75	2117.86	790.56	118.85	160.81	143.09
Max	3900	6984	6984	7910	12879	12879
Skewness	8.74	2.24	6.89	53.33	44.63	47.96
Skewness CI	[8.25, 15.22]	[1.87, 4.20]	[4.28, 10.76]	[41.08, 91.46]	[38.51, 80.44]	[38.32, 81.51]
Kurtosis	83.09	6.63	54.55	3425.56	3286.67	3544.64
Kurtosis CI	[76.42, 158.88]	[5.32, 12.06]	[13.03, 96.22]	[1121.99, 6417.24]	[2499.21, 6428.57]	[1912.13, 6685.30]
Sum	11264	12622	23886	235094	572962	808056

highly skewed (skewness represents the asymmetry of a distribution between its left and right probability tails). Notice that, in the *normal distribution*, skewness is equal to zero whereas kurtosis is equal to three, regardless of the variance.

Considering the large variations shown in Table V regarding the number of classes per project and branches per class, there is the question of whether using 110 projects is enough to obtain reliable results. As there is large variation, one could argue that even more projects should be used. However, there is a tradeoff between the number of software artifacts a researcher would like to use for an empirical study and the practical constraints (e.g., time and available hardware) of running large empirical studies. Even if using just 110 projects is acceptable, another approach to reduce the sampling error will be to first analyse *all* the Java projects in SourceForge (e.g., number of classes per project, and number of branches per class), and then use a stratified sampling technique (or one based on sample coverage scores like in [Nagappan et al. 2013]) instead of a random one to select 110 projects to use for experiments. However, depending on the factor of interest, this would be particularly time consuming, as each single project in SourceForge would need to be manually downloaded and compiled before it is amenable to analysis.

Another factor that could be considered when sampling from SourceForge is the *age* of the projects; for example, one could select only old projects, only new projects, or a balanced combination of different ages. For the type of research questions addressed in this paper, we did not consider age as an extra factor to consider in the experiments, and thus simply sampled projects independently of their age. Therefore, for future empirical studies where the age of a project might greatly influence the performance of an analysed technique, a new sample from SourceForge could be more appropriate than using the SF110 corpus.

Tables III and V report data for all the classes in SF110. However, there were 781 classes in these projects for which EVOSUITE did not output any result (493 of them in the `liferay` project alone). For many of these classes, the main reason is that loading them through reflection (i.e., `Class.forName`) fails, as an exception is thrown during execution of the static constructor of the class. Without being able to load the CUT, it is practically impossible to generate any test case for it. When a class is loaded, its static initializer is executed, but that can throw exceptions.

While debugging these exceptions in the static initializers, we found at least two different reasons. One is if the security manager denies some permissions. In a few cases, some static fields were initialized with method calls that led to the writing/creation of files, which are denied by our custom security manager (Section 2.1). For example, this happens when the static constructor of a class tries to create a log file. Such problems could be solved once techniques to handle writing of files are developed.

The second reason is more subtle. The initialization of a static field in the CUT could call a method on a static field in another class C that needs to be initialized first. Loading the CUT can result in a null pointer exception if the static initializer of C leaves its field set to null. The solution would be to first load C, call static methods (or access its static fields directly if they are public) to set the state in a consistent way, and then finally load the CUT, that now should not crash (i.e., `Class.forName` does not throw any exception). This is a particular example, as there can be other kinds of dependencies with other classes in the static initializer of the CUT. Unfortunately, automatically determining the reason for such exceptions in the static initializers, and solving them automatically before beginning to generate test cases for the CUT, is far from trivial. This problem is complex enough to warrant dedicated research on it (which was not possible for this paper).

Another reason, for which we obtained no data on some classes is that EVOSUITE crashed. This can happen if the JVM runs out of memory, which may happen as we do not have control on what the CUT might execute and allocate on the heap. EVOSUITE has several mechanisms to prevent this problem [Fraser and Arcuri 2013b], but they are not bullet proof (yet).

As these 781 missing classes represent only a relatively small fraction of the SF110 corpus, i.e., $781/23{,}886 < 3.26\%$, we do not believe they pose any particularly serious threat to the validity of this study, as in any case we are still using 23,105 classes for our analyses.

3.1.2. Industrial Software. Besides the experiments on SF110, we also carried out further experiments on industrial systems in one of our industrial partners' premises. One of the authors received access to the Java projects in the department of an external collaborator. This resulted in seven projects, totaling 4,208 classes. There were a further three projects, which we excluded from the analyses. Two of them were excluded for the following reason: those projects were compiled with Java 7, whereas on the engineer's machine only Java 6 was installed, and so their bytecode could not be loaded in the JVM running EVOSUITE. The third project was excluded due to compilation dependencies issues: for more than half of its classes, it was not possible to use reflection due to a `NoClassDefFoundError` caused by some missing libraries in the classpath. Although we had data for half of the remaining classes for that project we decided to exclude them because they could skew the final, overall results.

Table VI summarizes the properties of these industrial systems. Due to confidentiality restrictions, we can only provide minimal information on those systems. For example, all project names have been anonymized, and we are not allowed to name our industrial partner. Some of these systems are massive scale real-time controllers, others are data analysis tools with GUI front-ends. Note that Table VI reports data only for 3,970

Table VI: Details of the seven industrial projects. For each project, we report how many classes it is composed of, and the total number of bytecode branches.

Name	# Classes	# Branches
projectG	1,731	38,460
projectF	687	13,348
projectE	444	10,442
projectC	423	8,954
projectL	321	5,873
projectD	220	6,349
projectB	144	3,359
Total:	3,970	86,785

Table VII: Details of the artificially generated systems. For each project, we report how many classes it is composed of, and the total number of bytecode branches.

Name	# Classes	# Branches
tp1m	751	758,717
sp500k	301	307,546
tp10k	101	12,744
tp80k	81	61,560
tp50k	51	31,554
tp5k	31	2,850
tp7k	31	4,045
tp2k	21	1,041
tp1k	16	659
tp300	4	177
tp600	4	341
Total:	1,392	1,181,234

classes: there were 238 classes for which EVOSUITE generated no data (5% of the total; see above for discussion of the reasons).

3.1.3. Automatically Generated Software. Finally, in the literature there is also the idea of applying automatically generated software systems to reduce the bias of manual selection. In some domains (e.g., SAT solving) it seems to be standard to use generated examples; in unit test generation this is a relatively new idea, with tools like Rugrat [Hussain et al. 2012] only recently being introduced as potential alternative sources of software systems. Rugrat was used to produce artificial systems for the empirical study in the paper on the Carfast [Park et al. 2012] tool, and we use the systems from the Carfast paper[6] for further experiments and comparison. Table VII summarizes the properties of these software systems. Note that the Carfast paper mentions a further software system with about 1k LOC, which is not included in the archive on the website and therefore not part of our experiments.

3.2. Experiment Procedure

To answer the research questions posed in this paper, we first ran EVOSUITE on each of the 23,886 selected empirical study objects (i.e., classes) in SF110. To account for the randomness of the evolutionary search, we applied EVOSUITE 10 times on each employed study object with different random seeds and then studied the resulting probability distributions. Furthermore, we followed the guidelines in [Arcuri and Briand

[6]Available at: http://www.carfast.org, accessed June 2013

2012] to analyze all these data with appropriate statistical methods. In particular, we studied several statistics like mean, median, standard deviation, skewness and kurtosis. For all of them, confidence intervals were derived. Confidence intervals are a useful tool to statistically verify if a null hypothesis can be rejected (i.e., if it lies within the interval), and to compare different data sets (e.g., do their intervals overlap?).

In early experiments [Fraser and Arcuri 2013c], we applied EVOSUITE with a timeout of 10 minutes per each of the 1,741 classes used in the empirical study. As we apply the technique to a larger set of classes in this experiment (more than an order of magnitude larger), and a developer might not be willing to wait for 10 minutes to see a result, we chose a timeout of two minutes per class. This choice is based on our experience in achieving a reasonable trade-off between time and branch coverage. The search is ended after these two minutes, or as soon as 100% coverage was achieved.

The longer the search in EVOSUITE is left running, the better the results will be. However, EVOSUITE also has other phases with bounded computational time, like the building of the dependency graph of the CUT, minimization of the final test suite, and generation of JUnit assertions. Depending on the CUT, EVOSUITE might either spend most of its time in the search phase, but may also spend a non-negligible time in those other phases. Therefore, to be sure that the experiments finished within a predictable amount of time, we gave a general timeout of six minutes per class. In total, running all these experiments on SF110 took up to $(23{,}886 \times 10 \times 6)/(60 \times 24) = 995$ days (recall that, when 100% branch coverage was achieved, we stopped the search prematurely).

In all these experiments, we used EVOSUITE with its default parameter settings (e.g., population size and crossover probability). The effects of parameter tuning on search-based tools (e.g., EVOSUITE) has already been studied previously [Arcuri and Fraser 2013].

For the experiments using the 1,392 classes of the Carfast evaluation, we used the same procedure and configuration as for the experiments on open source projects, i.e., 10 runs, each with two minutes for test generation per class. On the 4,208 classes of the industrial systems, EVOSUITE was run only once, with the same settings as in the other experiments on open source software. More runs were not possible, as we were not permitted to take the bytecode of those projects and run the experiments on the cluster of computers we have access to. The employed machine was running Windows 7, with 12 cores (6 actual cores with hyper-threading enabled) at 3.07 GHz and 30GB of RAM.

To answer **RQ3** we used Randoop version 1.3.3[7], and applied it to each class with a timeout of one minute. On each class, Randoop was called with only that particular class specified as test class. We compiled the resulting test cases and used EVOSUITE to measure the branch coverage for each class.

3.3. Branch Coverage Results on SF110

Figure 2 shows the distribution of the average branch coverage results per project, for all the 110 projects in SF110. The average of the average branch coverage per project is 67%, where the project with lowest average coverage on its classes (*ipcalculator*) still has 20%. There are 30 projects for which the average branch coverage is above 80%. Within a certain degree, the distribution of coverage values in Figure 2 bears some resemblance to a normal distribution. Numerical statistics of these data are presented in Table VIII (together with the data results on the other industrial and artificial software systems). EVOSUITE on average created 3.8 tests per class, each having 8.2 statements on average (so on average 31 statements per test suite).

[7] https://code.google.com/p/randoop/, accessed October 2013

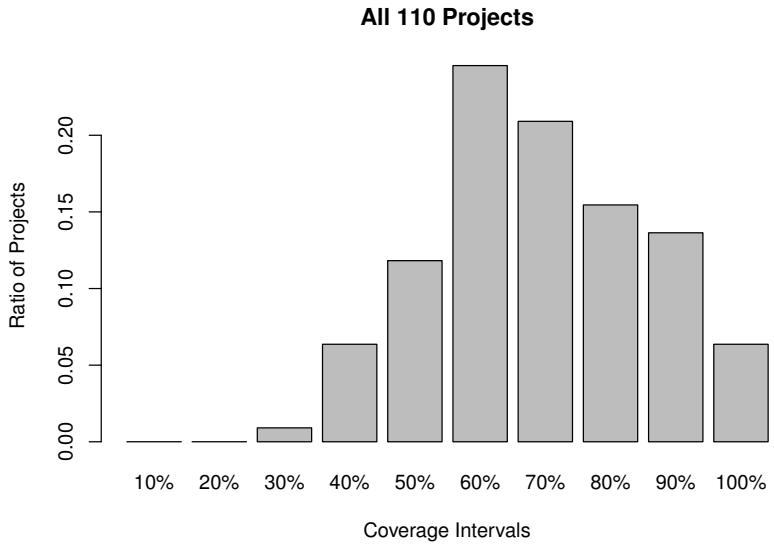


Fig. 2: Proportion of projects that have an average branch coverage (averaged out of 10 runs on all their classes) within each 10% branch coverage interval. Labels show the upper limit (inclusive). For example, the group 40% represents all the projects with average branch coverage greater than 30% and lower than or equal to 40%.

Table VIII: For each type of experiment, we present statistics on the obtained branch coverage. Results are grouped by class (average of its runs) and by project (average of all runs on its classes). Confidence intervals (CI) are at 95% level, and were calculated with bootstrapping.

Name	Grouping	Size	Min	Median	CI	Mean	CI	Std. Deviation	Max	Skewness	Kurtosis
All 110	Class	23105	0.00	0.94	[0.93, 0.95]	0.71	[0.70, 0.71]	0.37	1.00	-0.85	2.11
	Project	110	0.21	0.69	[0.66, 0.74]	0.67	[0.64, 0.71]	0.17	1.00	-0.24	2.57
Rand100	Class	11087	0.00	1.00	[1.00, 1.00]	0.78	[0.77, 0.78]	0.35	1.00	-1.26	2.98
	Project	100	0.21	0.69	[0.67, 0.75]	0.68	[0.65, 0.72]	0.17	1.00	-0.32	2.65
Top10	Class	12018	0.00	0.82	[0.80, 0.83]	0.64	[0.64, 0.65]	0.38	1.00	-0.56	1.69
	Project	10	0.32	0.55	[0.39, 0.61]	0.57	[0.49, 0.65]	0.14	0.77	-0.03	2.27
Industrial	Class	3970	0.00	1.00	[1.00, 1.00]	0.77	[0.76, 0.78]	0.35	1.00	-1.19	2.83
	Project	7	0.51	0.74	[0.62, 0.81]	0.75	[0.66, 0.86]	0.15	0.94	-0.39	2.16
CarFast	Class	1392	0.66	0.79	[0.79, 0.80]	0.81	[0.80, 0.81]	0.06	1.00	1.56	5.69
	Project	11	0.76	0.91	[0.88, 1.02]	0.87	[0.84, 0.91]	0.07	0.95	-0.55	1.73

EvoSuite produces test suites per class, and each project might have some difficult to cover and some easier to cover classes. Figure 3 therefore illustrates the distribution of branch coverage across the classes. This shows that there is a large number of classes which can easily be fully covered by EvoSuite (branch coverage 90%-100%), and also a non negligible number of classes with problems (branch coverage 0%-10%), while the rest is evenly distributed across the 10%-90% range. Although the average branch coverage per class is 71%, with such an extreme distribution presented in Figure 3, it might be misleading to only look at average values when speaking about classes in isolation.

The large number of classes with full branch coverage suggests that there are many classes that are trivially covered by EvoSuite. To analyze this further, Figure 4 illustrates, for each 10% branch coverage interval, the average number of branches

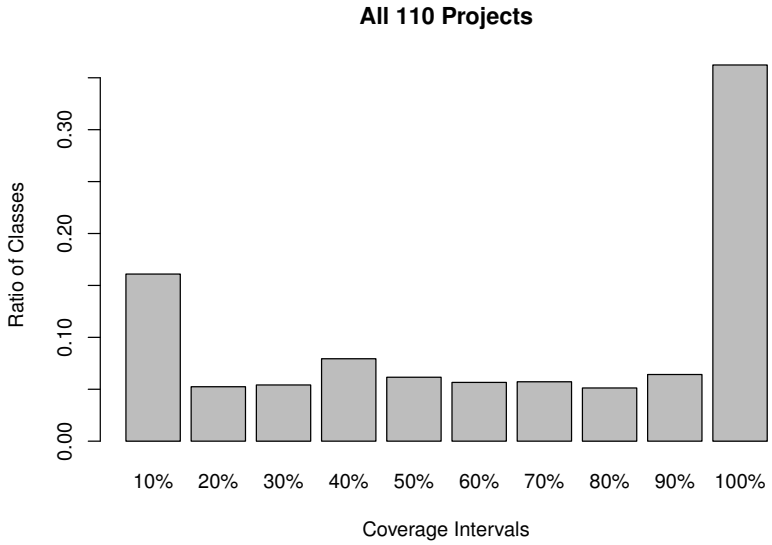


Fig. 3: Proportion of classes that have an average branch coverage (averaged out of 10 runs) within each 10% branch coverage interval. Labels show the upper limit (inclusive). For example, the group 40% represents all the classes with average branch coverage greater than 30% and lower than or equal to 40%.

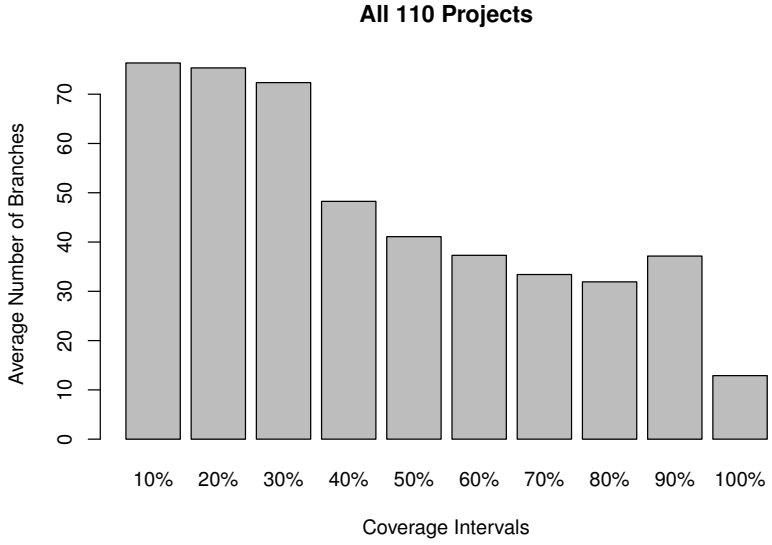


Fig. 4: Average number of branches of classes within each 10% branch coverage interval. Classes in the 90%-100% branch coverage range are the smallest, and thus potentially "easiest" classes.

Table IX: For each type of permission exception, we report the proportion of classes in which it is thrown at least once (first "Occurrence" column), and the average branch coverage for those classes. We also show the proportion of projects that have at least one class in which such an exception is thrown (second "Occurrence" column), and the average branch coverage for those projects (including all classes in those projects, regardless of what exceptions were thrown). Confidence intervals (CI) for the average branch coverage values are at the 95% level, and were calculated with bootstrapping.

Type	Per Class			Per Project		
	Occurrence	Mean Coverage	CI	Occurence	Mean Coverage	CI
No Exception	0.50	0.84	[0.84, 0.84]	0.055	0.83	[0.70, 1.03]
FilePermission	0.42	0.57	[0.57, 0.58]	0.95	0.66	[0.63, 0.70]
RuntimePermission	0.25	0.58	[0.58, 0.58]	0.92	0.67	[0.64, 0.70]
SocketPermission	0.30	0.51	[0.51, 0.52]	0.84	0.65	[0.62, 0.68]
SecurityPermission	0.066	0.42	[0.42, 0.43]	0.13	0.63	[0.55, 0.71]
AllPermission	0.0012	0.47	[0.44, 0.51]	0.12	0.64	[0.57, 0.70]
UnresolvedPermission	4.4e-06	0.75	[0.75, 0.75]	0.0091	0.74	[0.74, 0.74]
SQLPermission	3.1e-05	0.52	[0.48, 0.55]	0.018	0.75	[0.74, 0.76]
SSLPermission	5.7e-05	0.33	[0.20, 0.44]	0.0091	0.69	[0.69, 0.69]
AuthPermission	0.0017	0.52	[0.49, 0.55]	0.064	0.65	[0.55, 0.74]
Other Permissions	0.00099	0.45	[0.40, 0.49]	0.045	0.76	[0.64, 0.89]

of the classes within this interval. The 90%-100% interval contains on average the smallest classes, suggesting that a large number of classes are indeed easily coverable because they are very small. As one would expect, on average larger classes are more difficult to test.

RQ1: *On average,* EVOSUITE *achieves 71% branch coverage in two minutes, but there is extreme variation between "easy" and "difficult" to test classes.*

In general, the results on SF110 are similar to the results on our past experiments; for example, in [Fraser and Arcuri 2013c] EVOSUITE achieved 83% branch coverage on 20 hand-selected open source projects. However, what causes the 12% drop in branch coverage? The next research question investigates the influence of environmental dependencies on the achieved branch coverage.

3.4. Security Permission Results on SF110

In Figure 3, there is a large number of classes that apparently have problems (0%-10% branch coverage), and the question is what causes this. A possible reason for low branch coverage is if the tested classes try to execute unsafe code, such that the security manager prohibits execution.

To see to what extent this is indeed the case, Table IX lists the average branch coverage achieved for classes for each of the possible permissions that the security manager denied during the entire search (and not just in the final test suite given as output by EVOSUITE). Classes that raise no exceptions achieve an average branch coverage of 84%, whereas all classes that require some permission that is not granted have lower coverage. Consequently, interactions with the environment can be considered a prime source of problems in achieving branch coverage. It is striking that 42% of all classes led to a `FilePermission` exception that does not involve just reading — in other words, nearly half of all classes led to attempts to manipulate the filesystem in some way!

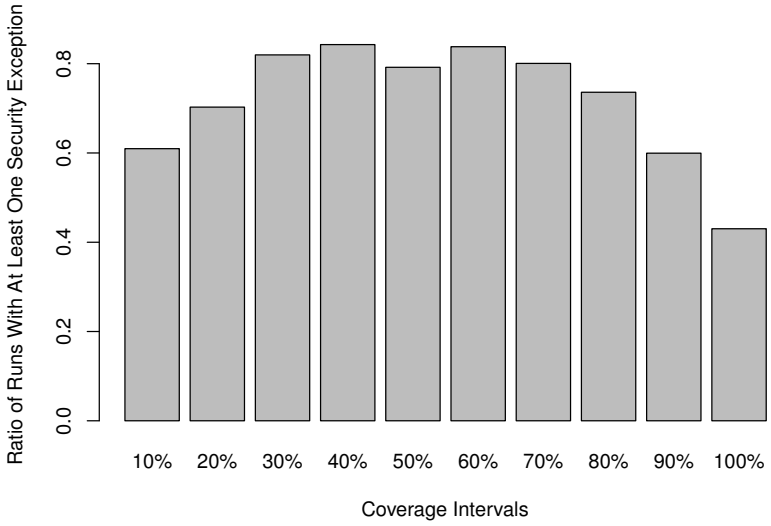


Fig. 5: Ratio of EVOSUITE runs for which there was at last one denied permission. Those ratios are grouped by 10% branch coverage intervals.

It is important to note that this I/O might not come directly from the CUT but one of its parameters: When testing object-oriented code one needs sequences of method calls, and as part of the evolutionary search EVOSUITE attempts to create various different types and calls many different methods on them. This means that just the existence of a denied `FilePermission` does not yet indicate a problem as there might be other ways to cover the target code that do not cause file access. Indeed Figure 5 shows that even classes that achieve high branch coverage often lead to some kind of denied permission check. However, the fact that classes with file access achieved significantly lower average branch coverage (57%) is a clear indication that file access *is* a real problem.

The other two dominant types of permissions we observed were `RuntimePermissions` (25% of classes in the SF110 corpus) and `SocketPermissions` (30%). `RuntimePermissions` can occur for various reasons, such as for example attempts to shut down the virtual machine or to access environment variables. As shown in Table I, we allowed several `RuntimePermission` operations that we deemed safe enough. But, unfortunately, the other operations we do not allow are still very common.

`SocketPermissions` happen when for example the CUT tries to open a TCP connection. It is not a surprise to see many of these exceptions in SF110, as the Java language is by construction well-suited for web applications, and several of the 110 projects are indeed web applications.

Finally, a common assumption for test generation tools is that the code under test is single-threaded, as multi-threaded code adds an additional level of difficulty to the testing problem. Creating a new thread does not require any permissions in Java; only terminating or changing running threads leads to permission checks. We therefore observed the number of running/waiting CUT threads each time any permission check was performed, each time a test execution timed out (EVOSUITE by default uses a timeout of five seconds per test case), and at the end of the test case execution.

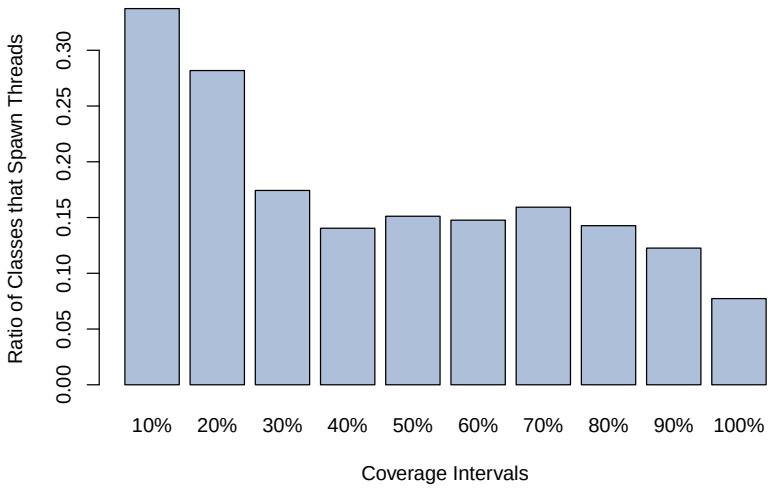


Fig. 6: Ratios of classes that spawn at least one thread for each 10% branch coverage interval.

Figure 6 illustrates the relation of branch coverage to the frequency of cases where we observed more than one thread: Classes that spawn threads usually obtain lower branch coverage. One reason is that, for the moment, EVOSUITE is not optimized to handle multi-threaded code. Although it can run classes that spawn threads (which is an essential requirement for using a testing tool on real-world software), the test case execution of the sequence of method calls is done on a single thread. If a method call on the CUT puts the test case executor thread on an object wait, then EVOSUITE would not be able to call a new method on a different thread that will wake up the executor.

Another problem is that, in the case of multi-threaded code, simply covering the code is usually not sufficient as test cases might become nondeterministic. Furthermore, multi-threading introduces new types of faults (e.g., deadlocks), and using a randomized algorithm (like EVOSUITE uses) on code that spawns new threads may cause problems, as Java offers no way to forcefully stop running threads. For this problem, EVOSUITE employs several advanced mechanisms to stop running threads (e.g., based on bytecode instrumentation), but they are not bullet proof [Fraser and Arcuri 2013b].

RQ2: *Multi-threading and interactions with the environment are very common problems that negatively affect branch coverage.*

The challenge will now be to overcome these problems, and to allow unit test generation tools like EVOSUITE to cover classes with environmental dependencies. One possibility, which we are currently investigating, is the use of *mocked* versions of core Java classes causing the interactions, such that the environment state is transformed into a test input that the test generation tool can explicitly set.

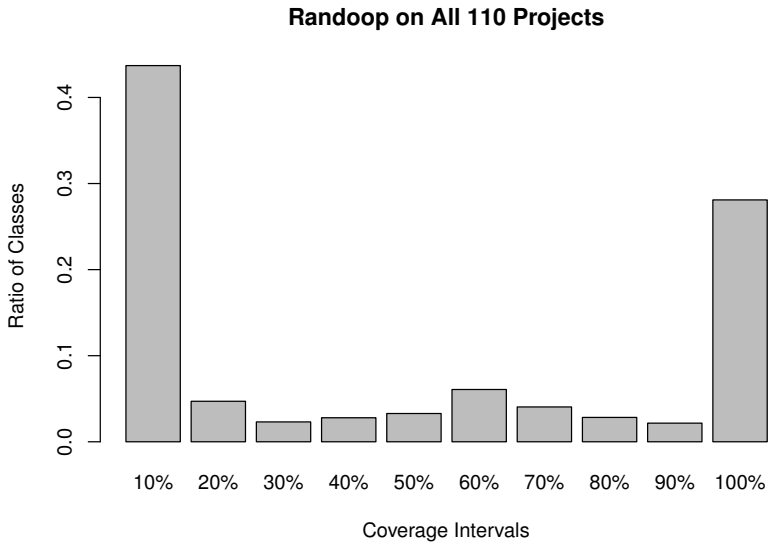


Fig. 7: Proportion of classes for which Randoop achieved branch coverage within each 10% branch coverage interval. Labels show the upper limit (inclusive).

Table X: For each type of permission exception found in the Randoop console output, we report how many times it is thrown in total and in how many classes it is thrown at least once.

Permission Exception	# of Occurences	# of Classes
com.liferay.portal.kernel.security.pacl.permission.PortalRuntimePermission	2,980,047	408
java.io.FilePermission	772,866	674
java.lang.RuntimePermission	85,810	50
java.net.SocketPermission	59,594	46
java.util.PropertyPermission	5,955	918
javax.xml.bind.JAXBPermission	3	1

3.5. Generalization to Other Test Generation Tools

Our experiments so far have shown that environmental dependencies affect the branch coverage that EvoSuite achieves. To verify whether this finding is specific to EvoSuite or applies to other test generation tools as well, we ran the Randoop [Pacheco et al. 2007] tool on all classes in SF110, by specifying one class at a time as input test classes to Randoop. The reason we chose Randoop out of all other tools (which we will discuss in Section 4.1, Table XII) is that it is fully automated (i.e., it does not require manually written test drivers or parameterized unit tests), it is popular and highly cited, freely available, and has been applied to many software systems in the past (e.g., 4,576 classes, see Table XII).

To avoid potential problems with file access, we launched Java with a custom security policy when running Randoop using `-Djava.security.manager -Djava.security.policy=<POLICY>`, as suggested by the authors of Randoop [Robinson et al. 2011]. Randoop is not released with a policy file, so we wrote one for it based on Table I. However, to measure branch coverage, we had to allow Randoop to write its generated test cases (i.e., we had to grant it writing permissions on a specific folder). In

general, the issue of distinguishing between interactions caused by the CUT and those caused by the testing tool is difficult, which is one among several technical reasons for why in EVOSUITE we had to implement a sophisticated, customised security manager.

As it would need more modifications to make a unit test generation tool suitable for experimentation on a cluster than just a security policy, we did not run the Randoop experiments on the cluster of machines we had access to. For example, if a test data generation tool is not able to properly mute the outputs (e.g., on the console) of the CUTs, it can easily take down an entire cluster by filling up all the hardrive space (e.g., if console outputs are automatically redirected by the job scheduler to text files) and overflow the intra-node network. These are experiences we drew from past experiments with earlier versions of EVOSUITE, much to the dismay of our cluster's administrators (and likely also other users). Some of the technical solutions to overcome these problems are discussed in [Fraser and Arcuri 2013b]. We therefore ran Randoop on a dedicated machine, but thus were limited to run it only once per CUT. Randoop was run with its default values, for 60 seconds per CUT. During an initial run where the policy did not prohibit execution of native code due to a mistake in the policy file, the runs of Randoop led to *deletion* of 49 out of the 110 projects in SF110 (i.e., their entire folders were wiped out). We then reran Randoop on those 49 projects.

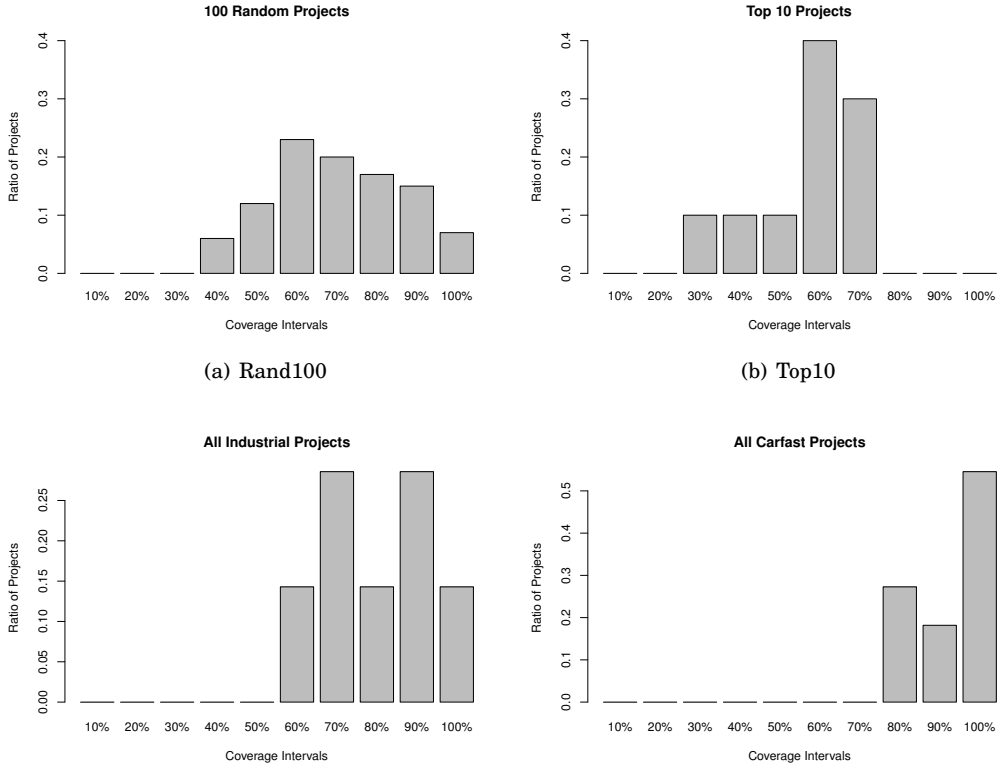
We compiled all the test cases generated by Randoop and used EVOSUITE to measure the branch coverage of these test cases for each class. As these runs did not use our custom security manager, we can only measure the number and type of security exceptions that propagate to the console output of Randoop during test generation.

Figure 7 summarizes the branch coverage achieved by Randoop. On average, Randoop achieves 40% branch coverage. As with EVOSUITE, the mean coverage is dominated by large sets of classes with low coverage ($<$10%) and simpler classes with high coverage ($>$90%). Table X summarizes the security exceptions that propagated to the console output of Randoop. There are fewer types of exceptions, and the number of classes with exceptions is also lower. To some extent, this is because not all security exceptions will actually propagate to the output. Furthermore, lower branch coverage can also reduce the chances of hitting code related to environmental interactions if such code is inside blocks that are not executed due to unsolved constraints (e.g., `if` statements with non-trivial predicates that would be hardly satisfied with random data). However, the number of exceptions is still high, providing evidence that our findings on environmental interactions are not specific to EVOSUITE.

RQ3: *Environmental interactions are not specific to* EVOSUITE *but also apply to other unit testing tools like Randoop.*

Note that our measurements are based on security exceptions. However, branch coverage will also be influenced by environmental interactions that do not lead to security exceptions. For example, a CUT might depend on the existence of a particular file, and in absence of that file might simply be uncoverable. This case would not show up in our statistics, as we permit file reading. Consequently, we conjecture that the problems caused by environmental dependencies on the achieved branch coverage are even more significant than our results can indicate.

The research field of automated unit test generation is often considered mature, but based on this result we are aware of no technique that practitioners could use *today* to automatically achieve high branch coverage on real-world software. For a successful technology transfer from academic research to industrial practice, it will be essential for the research community to solve all of these problems.



(a) Rand100

(b) Top10

(c) Industrial

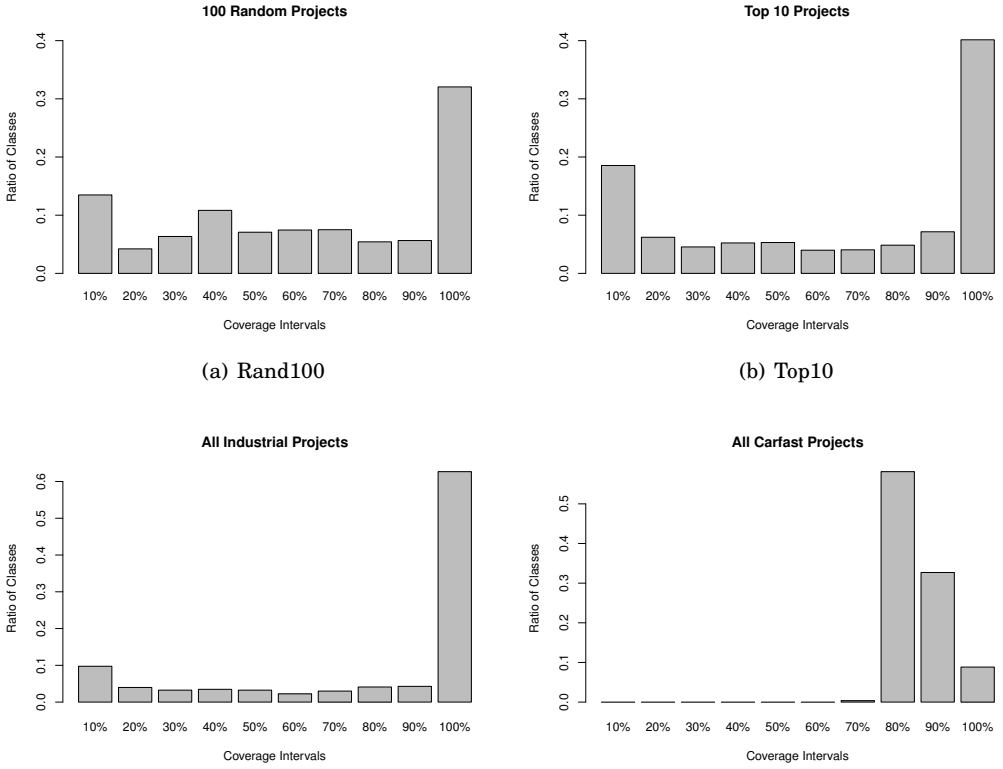
(d) Carfast

Fig. 8: Bar plots showing for each 10% branch coverage interval the proportion of projects that have an average coverage (averaged out of 10 runs on all their classes) within that interval. Labels show the upper limit (inclusive). For example, the group 40% represents all the projects with average coverage greater than 30% and lower than or equal to 40%.

3.6. Unbiased Selection vs. Top 10

The results so far considered the combination of projects chosen at random (e.g., *Rand100*) with the most popular ones (e.g., *Top10*). In this case, it is important to study whether their probability distributions related to achievable branch coverage differ or not. This is important if one wants to average and study results on all the employed software artifacts, without each time having to present data (e.g., with graphs and tables) separately for each group. Note that such properties would depend on the test case generation tool (e.g., EVOSUITE). Whether there would be only small differences, or not, among different testing tools is a matter that would require further empirical analyses (to this end, recall that SF110 is freely available).

Table V already showed results separately for *Rand100* and *Top10*, illustrating that the projects in *Top10* are usually bigger than the ones in *Rand100*. This is not surprising, as very small programs of just a few classes are not likely to be able to implement useful enough functionalities that millions of people would be interested in. Figure 8 shows the data that was presented in Figure 2, but by dividing them between *Rand100*



(a) Rand100

(b) Top10

(c) Industrial

(d) Carfast

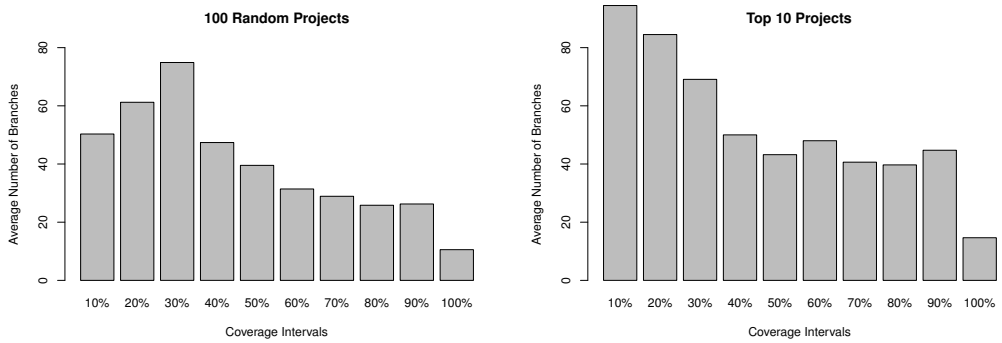
Fig. 9: Bar plots showing for each 10% branch coverage interval the proportion of classes that have an average branch coverage (averaged out of 10 runs) within that interval. Labels show the upper limit (inclusive). For example, the group 40% represents all the classes with average branch coverage greater than 30% and lower than or equal to 40%.

(Figure 8(a)) and *Top10* (Figure 8(b)). It is worth noting that, for *both* sets, most projects have average branch coverage between 50% and 90%.

Figure 9(a) and (Figure 9(b)) show the same data as Figure 3, but again by dividing them between *Rand100* and *Top10*. Surprisingly, the two sets present the same (or at least very similar) kind of irregular distribution. In other words, the largest group has over 90% branch coverage, and the second largest group is that with below 10% branch coverage, while the rest has similar occurrence.

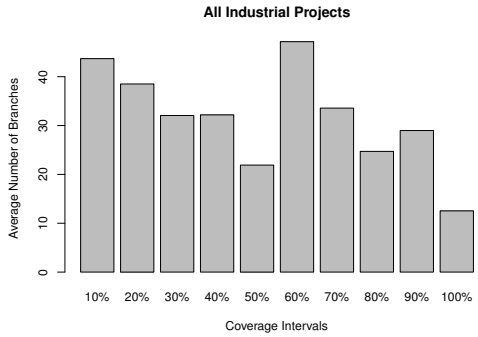
Figure 10 shows the same data as in Figure 4, but split between *Top10* and *Rand100*. In both cases, there is the same trend as it was in Figure 4: more complex classes (i.e., more bytecode branches) lead to lower branch coverage. This expected phenomenon is more marked for *Top10* than for *Rand100*, as visible in Figure 10.

Although *Rand100* and *Top10* projects show similar probability distributions, it is clear that *Top10* classes are significantly more difficult to test, i.e. average 64% branch coverage (*Top10*) compared to 78% (*Rand100*) (see Table VIII).

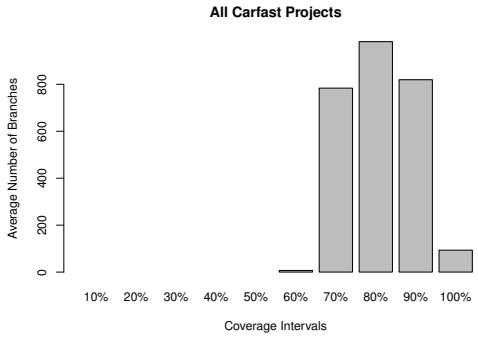


(a) Rand100

(b) Top10



(c) Industrial



(d) Carfast

Fig. 10: Bar plots showing for each 10% branch coverage interval, we report average number of branches of the classes with that average branch coverage.

RQ4*: In terms of achievable branch coverage with* EVOSUITE*, the top 10 most popular Java projects on SourceForge have similar characteristics with the other hosted projects, although they are considerably more difficult to test.*

The similarity between *Top10* and *Rand100* means that, for experiments in unit test generation, we can use the combined SF110 corpus of classes for representative empirical studies. However, adding *Top10* to *Rand100* makes the problem at hand more difficult, and so potentially underestimates the actual performance of a tool on an unbiased selection of projects like *Rand100*.

The similarity between the two sets (*Rand100* and *Top10*) was observed only for the problem domain addressed in this paper, and may not hold for other domains. For example, techniques that are dependent on the number of classes in a project would behave differently on the very large projects in *Top10* and the predominantly small projects in *Rand100*. Consequently, for problem domains other than unit test generation we can make no claims. However, although we still recommend using SF110,

it is also important to study *Rand100* and *Top10* separately, to see if there are any large differences between the two sets that needs to be further investigated.

3.7. Industrial Systems

With the results of the empirical analysis on the seven industrial systems, we want to provide data to help address **RQ5**. Similar to the experiments on open source software, Figure 8(c) shows average branch coverage per project, Figure 9(c) shows them by class, and Figure 10(c) shows them by numbers of branches. It is important to stress that looking at single classes in isolation would be statistically invalid in this case, as only one run of EvoSuite was carried out. But, when looking at properties on the entire projects, then there are enough data points (3,970) to reach some preliminary, meaningful results.

The average branch coverage values are higher for these industrial projects compared to SF110. Based on all the classes of the seven industrial projects, EvoSuite achieved an average of 77% branch coverage. However, when comparing the results of the industrial projects with just those of *Rand100*, obtained branch coverage is similar (77% and 78%). What is striking though is, that the distribution shapes in Figure 9(c) and in Figure 9(a) are very similar.

To some extent, these results came as a surprise to us: We were expecting to obtain (much) lower branch coverage values on these seven industrial projects, as they implement very complex functionalities. Can we conclude that industrial software is easier to test than open source software? Although these are seven real industrial systems, they do not represent a statistically valid sample of software developed in the whole industry, and thus we cannot draw any general conclusions. For example, there might have been some specific architectural decisions in these systems that are peculiar to the particular engineering department that developed them.

So what can we conclude from this experiment? There is reasonable evidence that EvoSuite works on software of this particular company as well as on open source software. The similarity between the results on SF110 and the industrial systems increases our trust that SF110 is a useful corpus for empirical studies.

> **RQ5**: *Experiments on seven real-world, industrial systems provide reasonable evidence that industrial code is not necessarily more difficult to cover for* EvoSuite *than open source software.*

Considering the limitations of running empirical studies on industrial software, it will be important to produce more evidence over time by running as many as possible new empirical studies and replications. However, it is unlikely there will ever be a "representative" benchmark for industrial systems (except maybe for very narrow subfields).

3.8. Automatically Generated Software

On the Carfast artificial software systems, EvoSuite achieved an average branch coverage of 80.6% (minimum 57.1%, median 79.4%). Similar to the other experiments, Figure 8(d) shows average branch coverage per project, Figure 9(d) shows it by class, and Figure 10(d) shows it by number of branches. Figure 8(d) suggests that all the individual projects seem similar in difficulty, and there are no projects with problems that EvoSuite is unable to handle; this is also confirmed by Figure 9(d). The difference between this set of artificial software and the other systems is probably most striking in Figure 10(d): The achieved branch coverage is generally high and does not seem to correlate with the number of branches.

Table XI: Pearson's product-moment correlation between the branch coverage achieved on each CUT (i.e., class) and different types of variables. Confidence intervals on the correlation values are at a 95% significance level.

Variable	Correlation	Confidence Interval
# of branches in the SUT	-0.17	[-0.17, -0.17]
# of classes in the project of the SUT	0.20	[0.19, 0.20]
Total # of branches in the project of the SUT	0.0052	[0.0011, 0.0093]
Average # of branches per class in the project of the SUT	-0.25	[-0.25, -0.25]

RQ6: EVOSUITE *achieves higher branch coverage on automatically generated software, as this software may not contain certain types of problematic classes.*

This difference in the results on open source software comes as a surprise, as a previous evaluation of Rugrat generated software [Hussain et al. 2012] suggested that software systems generated with Rugrat are very similar to open source software; it seems that software metrics are not necessarily good predictors of testability. To some degree this result is influenced by the absence of three categories of classes in these artificial systems: Trivially testable classes, classes with environmental dependencies, and "untestable" classes (i.e., badly designed classes where constructing parameter objects and reaching relevant states is very difficult). Arguably, these are categories of classes one would not want to use to demonstrate effectiveness of a technique that, for example, aims to improve the efficiency or scalability of a test generation approach. To some extent, the environmental dependencies aspect may be covered with a "Rugrat4Load" extension described on the Rugrat website, and the systems at hand are influenced by how they were configured for the Carfast experiments.

3.9. Effects of Class and Project Size

When choosing a set of software artifacts for software testing research, it is usually advisable to also include complex large systems to study the *scalability* of a proposed approach. Intuitively, one would expect a technique that works well on large, complex systems to also work well on smaller problems. Size is a very problem-specific property; in the case of unit test generation, we consider two dimensions: First, the size of an individual class for which unit tests are generated. Second, as empirical studies in unit testing are often based on selecting software projects and then including all their classes, we also consider the size of a software project. Table XI shows the correlations between branch coverage and these variables.

As one would expect, there is a weak negative correlation between the number of branches in the CUT and the average branch coverage that EVOSUITE achieved. This means that there is a tendency that on larger (measured in number of branches) CUTs, lower branch coverage is achieved given the same amount of search effort. Not only are there more testing targets (i.e., branches) to cover, but also each test case will take longer to run and evaluate during the search. The negative correlation between number of branches and branch coverage is not particularly strong, i.e., it is only -0.17. This is not a surprise, as (1) complex code (from the point of view of testing) involving network sockets and manipulating files can well be in small classes, (2) the whole test suite generation approach applied by EVOSUITE is known to alleviate the problem of

large classes [Fraser and Arcuri 2013c] and (3) in general, it is well known in the code complexity field that this kind of metrics have limitations [Weyuker 1988].

The second dimension of size we consider is the size of a project from which classes are taken. Larger classes have a tendency to be more difficult to test, so one might also expect that large projects involving thousands of classes will be more challenging. However, Table XI show a weak (0.2) *positive* correlation. This is a counter-intuitive result, as regardless of the strength of the effect size we would have expected a *negative* correlation (i.e., larger projects would be more difficult to unit test), not a positive one. This result holds only if we define the "size" of a project as the number of its classes, and not if we consider the sum of all the branches of all its classes (in which it seems there is no correlation). Note that we consider the difficulty of testing single classes, and not an entire project as a whole. Given the same testing budget per class, of course a project with more classes will obviously take longer to be unit tested.

Determining the exact reasons will be a matter of future research, but we conjecture that this result is due to the principles of object-oriented programming: A project that is developed following an adequate object-oriented methodology will tend to result in many small classes. Furthermore, it is likely that for an open source project to grow beyond a certain size it needs to be reasonably designed, otherwise it would be unlikely to attract sufficient developers in order to have it grow to that size. In contrast, small open source projects developed by single (potentially inexperienced) individuals may be abandoned after a short while.

One last correlation that we consider is the *average* number of branches in a project. In this case, there is a negative -0.25 correlation. Surprisingly, this correlation is stronger than the one with the number of branches in the CUT (-0.17). A possible explanation is that, when generating test cases for a particular CUT, there is also the need to initialise other objects that will be used as input to the methods of the CUT. If those other classes are particularly complex, then generating the right input data for the CUT will be more complicated.

RQ7*: There are only weak correlations between sizes and coverage. Larger classes tend to be more difficult to test with* EvoSuite*, but larger projects do not necessarily have more difficult classes.*

3.10. Threats to Validity

The main goal of this paper was to evaluate EvoSuite on different types of software in order to minimize the *threats to the external validity* of these experiments. However, there still remain some. The SF110 corpus is a statistically sound representative of open source projects, and our results are also statistically valid for other Java projects stored in SourceForge. For example, even if we encountered high kurtosis in the number of classes per project and branches per class, median values are not particularly affected by extreme outliers. To reduce this particular threat to validity, we used bootstrapping to create confidence intervals for some of the statistics (median, average, skewness and kurtosis). These confidence intervals allow the reader to judge how reliable the presented statistics and results are.

Our results might not extend to all open source projects, as other repositories (e.g., Google Code) *might* contain software with statistically different distribution properties (e.g., number of classes per project, difficulty of the software from the point of view of test data generation). Furthermore, there might be a significant percentage of open source projects that are not stored in any repository.

Furthermore, results on open source projects might not extend to software that is developed in industry, as for example financial and embedded systems might be under represented in open source repositories. To partially address this threat, we also carried out experiments on seven industrial systems, but this only serves as a sanity check: i.e., to see whether we get similar or very different results compared to open source projects. However, even if our results would be valid only for SourceForge projects, considering the two million subscribers of SourceForge they would still be of practical value and important for a large number of practitioners (both developers and final users).

The properties of the SF110 corpus are independent from which tool and technique was used for experimentation, but all conclusions based on coverage data are obviously dependent on the testing tool. In this paper, we used EvoSuite, and different tools might lead to different branch coverage results. As a sanity check, we also carried out comparisons with Randoop, a popular random testing tool. However, larger studies with other tools were not carried out for several reasons. Among the most obvious ones, there is the fact that often tools are not available, written for different programming languages (e.g., C#), intended for other usage scenarios (e.g., many test generation tools require explicit entry functions, which is infeasible to provide manually for SF110) or difficult to apply/adapt for experiments to run on a cluster. There can be several prototypes to compare with, and using all of them would be too time consuming (e.g., repeating the same experiments as those described in this paper on a different tool would have required another 995 days of computational resources), and so only some can be chosen — but then, similar to the choice of software artifacts, to avoid biased results, the choice of which tools/techniques to compare with should be done in a systematic way. An alternative, less biased, way to compare tools is through "tool competitions", where tools are run by the competition organizers, and the authors do not have access to the used benchmark (so as to avoid tuning on it). Such a competition for JUnit test data generation tools was for example held at the 6th International Workshop on Search-Based Software Testing (SBST 2013) [Bauersfeld et al. 2013]. EvoSuite participated [Fraser and Arcuri 2013a], and won by a large margin against the other tools. Therefore, we believe that EvoSuite can serve as a representative tool for Java unit test generation.

Threats to *internal validity* come from how experiments were carried out. We used the EvoSuite tool for our experiment, which is an advanced research prototype for Java test data generation. Although EvoSuite has been carefully tested, it might have internal faults that compromised the validity of the results. Because EvoSuite is based on randomized algorithms, we repeated each experiment on each class 10 times to take this randomness into account. Furthermore, the guidelines in [Arcuri and Briand 2012] were followed to properly analyze the statistical distributions of those runs.

A possible threat to *construct* validity is how we evaluated whether there are unsafe operations when testing a class. We considered the security exceptions thrown by all method calls in a test case, even when those methods do not belong to the class under test. Potentially, EvoSuite might have tried to satisfy parameters of the class under test using classes that lead to actions blocked by the security manager, even if these parameters could also have been satisfied with other classes that do not result in any security exceptions (e.g., when a method is declared to take an `Object` as parameter, EvoSuite considers every known class as a potential input).

A further threat to construct validity comes from the use of branch coverage. Even though reaching code is a necessity to finding bugs with testing, in practice other aspects might be important, such as the fault finding capability or the difficulty of manually evaluating the test cases for writing assert statements (i.e., checking the correctness of the outputs).

To verify whether the conclusions drawn from an empirical study are indeed correct, it is important to enable other researchers to independently replicate the performed study. To this end, both EVOSUITE and SF110 are freely available from our webpage [8].

4. CHOOSING SOFTWARE ARTIFACTS FOR EMPIRICAL STUDIES IN UNIT TESTING

In the previous section we described and analyzed an empirical study with EVOSUITE that is as sound as possible with respect to open source software, and a replication of the experiment on industrial and automatically generated software. At this point, it would be great to state what are the exact steps to follow in order to select a *perfect* set of artifacts for a desired subject of evaluation. However, the choice of software artifacts is typically subject to many constraints:

Availability. The main limitation when assembling a set of software systems is the availability of suitable subjects to choose from. Code-centric approaches have a clear advantage here, as they can leverage masses of available open source code.

Computational resources. A second limitation lies in the available resources to perform the experiment. A full experiment on SF110 requires the use of a cluster of computers, which is not available to all researchers.

Matureness of prototype. Empirical studies are typically performed using research prototypes, and research prototypes are inherently fragile and incomplete, as not many researchers are fortunate enough to have the resources to develop a prototype to a level where it can be applied to real-world programs.

Considering all this, the choice of software artifacts will always represent a trade-off between what is reasonable to demonstrate a technique works and the attempt to minimize the threats to validity of these experiments. Given the insights from our experiments using the EVOSUITE tool, we now discuss the potential implications of the choices that have to be made when assembling a set of software artifacts for an empirical study on unit test generation, or any other software engineering technique.

4.1. Common Practice in Unit Testing Experimentation

We start this section with a critical reflection of the current practice in software testing research, by surveying the literature on test generation for object-oriented software including studies. This is not meant to be an exhaustive and systematic survey, but rather a representative sample of the literature to motivate the need to reconsider the choice of software systems during empirical studies in software engineering research. Table XII lists the inspected papers and tools, together with statistics on their experiments.

We explicitly list how many out of the considered classes are container classes, if this was clearly specified. Container classes are an interesting category of classes that have particular properties that can be exploited for dedicated approaches for testing (e.g., [Boyapati et al. 2002]). Interestingly, 17 papers exclusively focus on container classes. We discuss implications of such a choice in Section 4.7. Note that some empirical studies use libraries such as Apache Commons Collections which are highly related to collections, but it is difficult to precisely quantify the containers; the table thus does not reflect the use of such projects.

We also list whether the evaluation classes were selected from open source code, industrial software, the literature, or were constructed for that evaluation. For industrial code, there often is no choice, because the systems are selected and provided by an industrial partner. Out of 50 evaluations we considered in our literature survey, 34 selected their software artifacts from open source programs, while only six evaluations included industrial code. This is to be expected, as it is difficult to get access to industrial

[8] http://www.evosuite.org/SF110

Table XII: Evaluation settings in the literature. The container column denotes how many of the classes are container data structures, in those cases where this was determinable. The source column describes whether software artifacts were chosen from available open source projects (OS), industry projects, taken from the literature, or created by the authors.

Tool	Reference	Projects	Classes	Containers	Source
APex	[Jamrozik et al. 2012]	9	9	0	Open Source
Artoo	[Ciupa et al. 2008a]	1	8	8	Open Source
AutoTest	[Ciupa et al. 2008b]	1	27	17	Open Source
Ballerina	[Nistor et al. 2012]	6	14	4	Open Source
CarFast	[Park et al. 2012]	12	1,500	-	Generated
Check'n'Crash	[Csallner and Smaragdakis 2005]	2	-	1	OS / Literature
Covana	[Xiao et al. 2011]	2	388	-	Open Source
CSBT	[Sakti et al. 2012]	2	3	2	Open Source
DiffGen	[Taneja and Xie 2008]	1	21	8	Literature
DSDCrasher	[Csallner et al. 2008]	2	24	-	Open Source
DyGen	[Thummalapenta et al. 2010]	10	5,757	-	Industrial
Eclat	[Pacheco and Ernst 2005]	7	631	16	OS/Lit./Constr.
eCrash	[Ribeiro et al. 2010]	1	2	2	Open Source
eCrash	[Ribeiro et al. 2009]	1	2	2	Open Source
eToc	[Tonella 2004]	1	6	6	Open Source
eToc	[McMinn et al. 2012]	10	20	0	Open Source
EvaCon	[Inkumsah and Xie 2008]	1	6	6	Open Source
EvoSuite	[Fraser and Arcuri 2011a]	6	727	-	OS + Industrial
EvoSuite	[Fraser and Arcuri 2013c]	20	1,741	-	OS + Industrial
Jartege	[Oriat 2005]	1	1	-	Constructed
JAUT	[Charreteur and Gotlieb 2010]	3	7	-	Constructed
JCrasher	[Csallner and Smaragdakis 2004]	1	8	2	Literature
JCute	[Sen and Agha 2006]	1	6	6	Open Source
jFuzz	[Karthick Jayaraman and Kiezun 2009]	1	-	-	Open Source
JPF	[Visser et al. 2004]	1	1	1	Open Source
JPF	[Visser et al. 2006]	1	4	4	Constructed
JTest+Daikon	[Xie and Notkin 2006]	1	9	9	Constructed / Lit.
JWalk	[Simons 2007]	6	13	-	Constructed
Korat	[Boyapati et al. 2002]	1	6	6	Literature
MSeqGen	[Thummalapenta et al. 2009]	2	450	-	Open Source
MuTest	[Fraser and Zeller 2012]	10	952	-	Open Source
NightHawk	[Andrews et al. 2007]	2	20	20	Literature
NightHawk	[Andrews et al. 2011]	1	34	34	Open Source
NightHawk	[Beyene and Andrews 2012]	2	-	-	Open Source
OCAT	[Jaygarl et al. 2010]	3	529	-	Open Source
Palus	[Zhang et al. 2011]	6	4,664	-	OS + Industrial
Pex	[Tillmann and Schulte 2005]	2	8	-	Constructed
PexMutator	[Zhang et al. 2010]	1	5	1	Open Source
Randoop	[Pacheco et al. 2007]	14	4,576	-	OS / Industrial
Rostra	[Xie et al. 2004]	1	11	9	Constructed / Lit.
RuteJ	[Andrews et al. 2006]	1	1	1	Open Source
Symclat	[d'Amorim et al. 2006]	5	16	12	Constructed / Lit.
Symstra	[Xie et al. 2005]	1	7	7	Literature
Symbolic JPF	[Păsăreanu et al. 2008]	1	1	-	Industrial
Symbolic JPF	[Staats and Pasareanu 2010]	6	6	4	Industrial/OS
TACO	[Galeotti et al. 2010]	6	6	6	OS/Lit.
Testera	[Marinov and Khurshid 2001]	4	4	2	Open Source
TestFul	[Baresi et al. 2010]	4	15	12	OS + Literature
YETI	[Oriol 2012]	100	6,410	-	Open Source
N/A	[Arcuri and Yao 2008]	1	7	7	Open Source
N/A	[Wappler and Wegener 2006]	2	4	4	Open Source
N/A	[Andrews et al. 2008]	2	2	1	Open Source

code, and even if one gets access it is not always easy to publish results achieved on this code due to privacy and confidentiality issues. We also include the .NET libraries as industrial code here, although the bytecode is available freely. On the other hand, 17 evaluations used artificially created examples, either by generating them or by reusing them from the literature.

If it is not justified how a particular set of classes was selected for evaluation, then in principle it could mean that the presented set represents the entire set of classes on which the particular tool was ever tried, but it could also mean that it is a subset on which the tool performs particularly well. Interestingly, only a single paper [Oriol 2012] out of those considered justifies why this particular set of classes was selected, and how this selection was done. In his study, Oriol [Oriol 2012] used the tool YETI to evaluate some general laws of random testing. The software artifacts were taken from the Qualitas Corpus [Tempero et al. 2010], from which 100 classes were chosen at random from each of the projects in that corpus. The Qualitas Corpus [Tempero et al. 2010] is a set of open source Java programs that were originally collected to help empirical studies on static analysis. This corpus has been extended throughout the years, although it is not clear whether any formal criterion was applied to choose which projects to include (i.e., although such corpus features many kinds of different software applications, their choice seems manual).

4.2. Size vs. Statistical Power

With an automated tool like EvoSuite, it could be possible to have even larger (in terms of the number of artifacts considered) empirical studies than those presented in this paper. There is no particular technical problem that prevented us from using $20{,}000$ projects and repeating each experiment $1{,}000$ times — it all depends on available resources. As it was, our experiments on SF110 took 995 days of computational effort. This would make such an experiment impossible, unless a cluster of computers is available. In some cases, it is not only a matter of time, but rather of resources. For example, for their experiments on the CarFast tool, Park et al. [Park et al. 2012] used the Amazon EC2 virtual machine. An empirical study estimated to take 1,440 computational days, resulted in a US$30,000 cost.

For a fixed amount of computational resources (e.g., 1000 days on a cluster), there is a tradeoff between the number of used artifacts in the empirical study and repeated runs with different seeds. As explained in more detail in [Arcuri and Briand 2012], one needs to strike a balance between threats to external validity (e.g., number of artifacts) and statistical power (e.g., number of runs). This balance depends on the addressed research questions, and as such it will vary from study to study. For example, if a researcher wants to compare two testing techniques, and check on which classes one is better than the other, then it would make more sense to have fewer software artifacts (e.g., 100 or $1{,}000$ classes chosen at random from *Rand100*) in order to have a higher number of runs (within the same budget for computational resources). More runs would be helpful to have enough statistical power (to detect statistical differences, if any) when the two techniques are compared on a per class basis. However, sampling classes randomly from *Rand100* is not the same as sampling directly from SourceForge, as *Rand100* itself is a sample of SourceForge which is subjected to sampling error.

Before running an experiment, in general one would not know what is the magnitude that the effect sizes will have. Therefore, choosing the right number of runs for an experiment to obtain enough statistical power is cumbersome. One approach would be to first run an experiment with a pre-defined number of runs, calculate the effect size, and then apply *power analysis* techniques [Cohen 1988]. Power analysis can give an indication of how many more runs (i.e., data points) one would need to achieve the desired level of statistical significance given the obtained effect size. After such an

analysis, more runs can be carried out and new statistical tests can be performed. For a more in details discussion on these points, we refer to [Arcuri and Briand 2012].

4.3. Does Size Solve All Problems?

Assuming one has sufficient resources to run a large experiment with sufficient statistical power, does using a large and variegated set of software artifacts solve the problem of external validity? The answer is unfortunately *no*. For example, let us critically look at our own previous work on EvoSuite. When we first presented the whole test suite approach in [Fraser and Arcuri 2011a], the empirical study was composed of five libraries (java.util, Joda Time, Commons Primitives, Commons Collections and Google Collections) and a small package from an industrial system, consisting of a total of 727 public classes. In the successive extension [Fraser and Arcuri 2013c], we used 19 libraries and the same industrial package, totaling 1,741 public classes. On these software artifacts, EvoSuite obtained 83% branch coverage on average, which is higher than the 71% reported in this paper on SF110. The selection of the six (for [Fraser and Arcuri 2011a]) and 20 (for [Fraser and Arcuri 2013c]) projects was manual. Although we tried to strike a balance between different types of libraries, the selection was still biased. But, even if the selection was not biased, are such numbers of projects enough for an empirical study? We have shown in Table V that there are extreme variations among projects, so one should try to have a high number of projects in an empirical study. If six and 20 projects were chosen from SF110, what would be the best possible and the worst possible result for a study that size?

For each of the 110 projects in SF110, we have an average (on their classes) branch coverage value. On one hand, if we choose the six projects from SF110 with lowest average branch coverage, the average on those six projects would be only 30%. On the other hand, if we choose the ones with the highest branch coverage, the average on those six would be 97%. The difference between a "lucky" choice of six projects for an empirical study and an "unfortunate" one can lead to branch coverage values that are different by 67%! If instead of six projects we take 20, then a lucky choice would give 91% branch coverage, whereas an unfortunate one would result in 41% branch coverage. The difference is this time lower (50%), but still very high. Our own experiments in [Fraser and Arcuri 2013c] resulted in 83% branch coverage — and in retrospective, we probably did select libraries devoid of environmental dependencies.

In general, there is so much variation in average branch coverage values among different projects that, considering these data, the choice of which projects to use in an empirical study is an important decision that needs to be done deliberately and systematically. A further issue that our analyses show is that, only concentrating the research effort on large projects may bias results (see correlation analysis in Table XI), as smaller projects showed a tendency towards being more difficult to test in our experiments. This suggests that having few large projects may be less preferable than using many smaller ones, although the correlations in Table XI are relatively weak.

Furthermore, the results on **RQ7** (Section 3.9) suggest that, if one wants to somehow quantify the difficulty/complexity of a project (e.g., when choosing a set of artifacts for experimentation) by considering the number of classes or branches in it, this would be misleading for unit testing. A more appropriate measure might be to consider the average number of branches per class, or use some other code complexity metric. For example, Daniel and Boshernitsan [Daniel and Boshernitsan 2008] trained a decision tree based on several code metrics to estimate the difficulty of a class before a test data generation tool is applied to it.

4.4. Industrial vs. Open Source

Although open source software is widely used, it only represents one face of software development. In all likelihood, there are many more industrial programs than open source programs. Because industrial software might be very different from what can be found in open source repositories, it would be important to evaluate testing techniques on software in industry. However, it is the sad truth and the topic of many a panel discussion these days that accessing industrial software is difficult for researchers.

Even when it is possible to engage with industrial partners for experiments, there are issues related to the properties of such industrial systems. First, the empirical study would be biased and those employed systems cannot represent a statistically valid sample, as their choice is usually limited to the industries one has contacts with (e.g., located in the same area, city). Any generalization attempt from a biased selection is limited. Second, in most cases an empirical analysis based on industrial software is not replicable by other researchers, usually due to confidentiality restrictions (e.g., software companies are usually not willing to share their intellectual property, as it represents critical assets). Replication, however, is a critically important aspect of software engineering experimentation. Besides replication, a common point of reference to compare with (e.g., SF110) will be essential for meta-analyses when conclusions are inferred by studying and combining the data of different empirical studies (and so different industrial systems). Consequently, we argue that when possible, experiments using open source software are preferable.

Of course, experiments on industrial software are still very important, and should be done whenever possible. Indeed, precisely *because* obtaining data from industry is very challenging, we strongly believe that using industrial systems and comparing them to SF110 or *Rand100* will be essential to obtain over time a reliable body of empirical evidence.

However, one should be aware of the limits of how far results can generalize. In this regard, the results of **RQ5** are of high practical value for researchers in software testing. As long as there are no new experimental results showing differing performance of testing tools between SF110 and industrial systems, it can be safe to assume that good results achieved on SF110 can be of value for practitioners in industry. This is especially important for researchers who are not in contact with industrial partners, and can only do empirical studies on open source software.

4.5. Real vs. Generated Software

When generating software automatically, the bias introduced when selecting projects manually is avoided entirely. However, bias may be introduced through (a) the capabilities of the benchmark generation tool and (b) the particular configuration applied when running the tool. Consequently, automatically generated software is not free of bias either: Automatically generating software removes the selection bias, but adds bias in terms of capabilities and configuration of the code generation tool.

A particular strength of automatically generating software is that the type of problem generated can directly be influenced. In the case of the Carfast study, the particular targeted problem seems to be largely independent classes without environmental dependencies, but high code complexity. Indeed, the artificial software systems studied seem to be good for this (e.g., the proportion of classes with 100% branch coverage achieved by EvoSuite is much smaller than for the other types of software). Consequently, automatically generated software seems well suited to evaluate particular techniques or optimizations. However, it seems that, despite static similarities to open source software, automatically generated software does not yet serve as a replacement for "real" programs. When for a given domain there are no artifacts to choose from available

on the internet, then automatically generating them may be the only option. In general, however, to draw conclusions about how results generalize, experiments on open source programs are still preferable.

4.6. Effectiveness vs. Practical Relevance

An important point in choosing a set of artifacts for an empirical study lies in the objective of the planned experiments. Is it reasonable to look for sound and unbiased studies in all research papers in the future? The answer is clearly no. In many cases this is not feasible, and sometimes it is not even desirable. For a technique X that addresses a specific problem, that particular problem may only have few instances in a corpus like SF110 or *Rand100*. Even if the given technique applied to the targeted problem in isolation can have strong results, applying it to *Rand100* may diffuse that effect. If the problem addressed by the technique is not common in *Rand100*, then the measurable effect may be very small if other types of problems are not affected by the technique. However, the opposite is also possible: For example, software modularization is commonly applied to large software projects, of which there are few in *Rand100*. Thus, measuring the performance of a modularization technique on *Rand100* may lead to very optimistic results, as the performance on small projects is likely very good for *any* modularization technique. Such results would be valid, but of less practical value as one would not apply such a technique on small projects in the first place.

Consequently, we note that *Rand100* is a representative sample of open source software that is well suited to answer questions such as:

— Does technique X have wide scope? I.e., how common is the addressed problem?
— Having shown that X overcomes a specific problem, are there any negative side-effects on other software not affected by that problem?

Ideally, one would still perform an unbiased selection (e.g., by "filtering" a relevant subset of *Rand100*), and then run on something like *Rand100* to check for negative side effects.

4.7. Focused Studies: Container Classes

An example of a specific type of classes addressed in the literature on unit test generation are container classes. As already mentioned in Section 4.1 (e.g., see Table XII), many empirical studies are exclusively based on container classes (e.g., vectors and lists). When addressing a problem as specific as container classes, a large and unbiased corpus such as SF110 would not be suitable: Only few of the classes in SF110 are container classes, and among the 4,208 classes in the employed industrial software, not a single one was a container class. Any effects of a technique that specifically improves testing of container classes would be practically insignificant when evaluated on an unbiased set of software systems, regardless of how effective it would be for container classes.

However, while it is certainly true that containers are widely used, writing new container classes may not be such a common scenario (e.g., considering the existence of mature libraries such as java.util). Consequently, if a technique targets unit test generation in *general*, then focusing on container classes may create a strong bias in the results. For example, containers do not have GUIs, they do not spawn threads, they do not require sequences of function calls to set the internal states of their inputs (e.g., one can insert/remove integers), they do not write to disk, they do not open TCP connections and they usually do not have constraints based on string matching and float arithmetic. We observed in our experiments on SF110 and the industrial projects that these are common characteristics of classes developed in industry and in open source projects,

therefore these problems would likely outweigh any improvements specific to container classes.

Does using a larger set of software artifacts automatically avoid this problem? Let us take another critical look at our initial EVOSUITE study [Fraser and Arcuri 2011a], where we used five libraries and an industrial system, resulting in a total of 727 public classes. At first look, this sounds like a reasonable choice. A closer look reveals that among the libraries we included java.util, Apache Commons Primitives, Apache Commons Collections, and Google Collections. In other words, four out of the six projects provide different types of containers and helper classes. This may well be a contributing factor to the high average branch coverage achieved in these experiments. Indeed there is evidence that high coverage on container classes is even possible using random testing [Sharma et al. 2011].

Consequently, if a testing tool aims to improve unit testing in *general*, then regardless of its size the employed set of artifacts should aim to cover a representative variety of software subjects, rather than focusing on one particular type (e.g., container classes).

4.8. Libraries vs. Applications

In the followup experiment [Fraser and Arcuri 2013c] to the original EVOSUITE experiments [Fraser and Arcuri 2011a], we added 14 additional projects to our empirical study, yet the branch coverage remained as high as 83% on average. Although we tried to be systematic and unbiased in the selection, these projects all shared similarities that are not prevalent in SF110: All 19 open source projects in [Fraser and Arcuri 2013c] were *libraries*.

We conjecture that libraries in general are "easier" for unit testing: They provide extensive public APIs, these APIs have evolved over the years based on extensive use in other projects, and they are often surprisingly well tested, again suggesting that the APIs are well-designed with testing in mind. In contrast, many of the SF110 projects are not libraries but *programs*, most of them without unit tests. Unlike libraries, programs do need to offer rich APIs, and often there are complex dependencies between the components of a program, whereas libraries often consist of independent or only loosely coupled units. The distinction between libraries and actual applications is therefore something to pay attention to when software artifacts are manually selected. Likely this observation is specific to unit testing; however, other testing domains will likely have similar categories of subjects one might need to be aware of.

4.9. Technical Limitations

The experiments performed in this paper show that 50% of the classes may lead to interactions with the environment that inhibit high branch coverage in the best case, and are *potentially harmful* in the worst case. When there are no interactions, average branch coverage is as high as 84%, which is higher than the average 71%. The environment problem is not a new discovery – in manual testing, mocking frameworks are part of the standard repertoire of a tester. For Java, there are many popular frameworks (e.g., Mockito[9], EasyMock[10] or JMock[11]). The use of mocking frameworks within automated test generation tools is less common. An example is the Moles framework [de Halleux and Tillmann 2010], which has been used in conjunction with the Pex tool [Tillmann and Schulte 2005]. While representing a promising approach, the lack of extensive evidence (the empirical study in [de Halleux and Tillmann 2010]

[9]http://code.google.com/p/mockito/, accessed June 2014
[10]http://www.easymock.org, accessed June 2014
[11]http://jmock.org, accessed June 2014

was based only on a 15 lines long method) suggests that there are still opportunities for further research.

Besides highlighting the importance of addressing this type of engineering problems in testing tools, the existence of environment interactions represents an important consideration when choosing empirical study subjects. In particular, when studies are repeated and comparative experiments with different tools are performed, it is important to be clear about technical limitations influencing the results and the choice of employed software artifacts.

5. CONCLUSIONS

Like all experimentation in software engineering research, our past empirical investigations of the EVOSUITE unit test generation tool inherently suffer from a common threat to external validity, caused by the choice of software artifacts for experimentation. To study how our past findings on EVOSUITE generalize, and to analyze the effects of the choice of software artifacts in our empirical studies, in this paper we introduced the SF110 corpus: 110 open source projects consisting of 23,886 Java classes, for a total of more than 800 thousand bytecode branches and more than 6.6 millions of lines of code. We ran experiments with the EVOSUITE tool, which automatically generates test cases aiming at maximizing branch coverage. To the best of our knowledge (see Section 4.1), this corpus does not only represent one of the largest sets of software systems in the literature of empirical studies on test data generation for object-oriented software to date, but most importantly it helps to reduce the threats to external validity of the experiments we ran. This is important as external validity is one of the main barriers for a successful transfer of research results to software development practices.

We applied our research prototype EVOSUITE on this statistically valid corpus, as well as seven industrial systems and artificial software. Besides confirming good levels of achieved branch coverage, these experiments allowed us to investigate the effects of the choice of software artifacts on the branch coverage achieved by EVOSUITE. Comparisons with past results on EVOSUITE clearly demonstrate the significance of a proper choice of software artifact: Even a selection of subjects that would be considered as large can more or less produce any result, depending on the software projects selected for the empirical analyses. Even if the "perfect" selection of software artifacts does not exist for practical reasons, it is therefore important to be systematic and clear in the choice of the subjects employed in an empirical study.

Besides the experiments on the SF110 corpus, in this paper we also used seven industrial systems. Because in contrast to SF110 they are a biased sample, no general conclusion can be derived from their analysis. However, as obtaining real data from industry is very challenging, we believe that reporting case studies on industrial software, and comparing them to SF110, is important to obtain a reliable body of empirical evidence over time.

Bias in the software artifacts selection can also cause important practical limitations to be obfuscated behind positive results. In the case of unit test generation, our experiments revealed that the large majority of classes (i.e., 50%) may lead to potentially unsafe interactions with the environment, thus inhibiting branch coverage, creating dependencies between tests, and in the worst case harming the execution environment. On classes without unsafe operations, EVOSUITE achieves on average an 84% branch coverage, while on the entire SF110 corpus the average coverage is only 71%. On classes manipulating the file system, branch coverage goes down to 57%, and down to 51% for the classes using network sockets. Environment interactions are not only an issue for EVOSUITE, but also for other tools, as for example witnessed by experiments using Randoop.

Our experiments in this paper focused on the Java programming language and the EVOSUITE unit test generation tool. However, the problem of choosing a proper set of software artifacts for experimentation is independent of the programming language and the addressed software engineering task. For other programming languages, corpora that are similar to SF110 can be created. For example, the evaluation of a tool for C# can be based on choosing 100 C# projects at random from SourceForge, plus the top 10 most popular C# projects. Furthermore, beside automated unit test generation, SF110 can also be used for empirical analyses in other software engineering tasks, like for example mutation and regression testing.

For more information on EVOSUITE and the SF110 corpus of classes, please visit our website at:

`http://www.evosuite.org/SF110`

ACKNOWLEDGMENTS

We would like to thank Sai Zhang for help in setting up Randoop to run on SF110.

REFERENCES

J. H. Andrews, A. Groce, M. Weston, and R. G. Xu. 2008. Random Test Run Length and Effectiveness. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. 19–28.

James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. 2006. Tool support for randomized unit testing. In *Proc. of the 1st Int. Workshop on Random Testing (RT '06)*. ACM, New York, NY, USA, 36–45.

James H. Andrews, Felix C. H. Li, and Tim Menzies. 2007. Nighthawk: a two-level genetic-random unit test data generator. In *Proceedings of the 22nd IEEE/ACM Int. Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 144–153.

J. H. Andrews, T. Menzies, and F. C.H. Li. 2011. Genetic Algorithms for Randomized Unit Testing. *IEEE Transactions on Software Engineering (TSE)* 37, 1 (2011), 80–94.

A. Arcuri and L. Briand. 2012. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* (2012). DOI: 10.1002/stvr.1486.

Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)* (2013), 1–30. DOI: 10.1007/s10664-013-9249-9.

A. Arcuri, M. Z. Iqbal, and L. Briand. 2012. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering (TSE)* 38, 2 (2012), 258–277.

Andrea Arcuri and Xin Yao. 2008. Search based software testing of object-oriented containers. *Inform. Sciences* 178, 15 (2008), 3075–3095.

L. Baresi, P. L. Lanzi, and M. Miraz. 2010. TestFul: an Evolutionary Test Approach for Java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 185–194.

S. Bauersfeld, T. Vos, K. Lakhotia, S. Poulding, and N. Condori. 2013. Unit Testing Tool Competition. In *International Workshop on Search-Based Software Testing (SBST)*. 414–420.

M. Beyene and J.H. Andrews. 2012. Generating String Test Data for Code Coverage. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 270–279.

Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, USA, 123–133.

Florence Charreteur and Arnaud Gotlieb. 2010. Constraint-Based Test Input Generation for Java Bytecode. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE '10)*. IEEE Computer Society, Washington, DC, USA, 131–140.

I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. 2008a. ARTOO: adaptive random testing for object-oriented software. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 71–80.

I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. 2008b. On the Predictability of Random Tests for Object-Oriented Software. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 72–81.

J. Cohen. 1988. Statistical power analysis for the behavioral sciences. (1988).

Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.* 34 (2004), 1025–1050. Issue 11. DOI:http://dx.doi.org/10.1002/spe.602

Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*. ACM, New York, NY, USA, 422–431.

Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.* 17, Article 8 (May 2008), 37 pages. Issue 2.

Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. 2006. An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. 59–68.

Brett Daniel and Marat Boshernitsan. 2008. Predicting effectiveness of automatic testing tools. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 363–366.

Jonathan de Halleux and Nikolai Tillmann. 2010. Moles: tool-assisted environment isolation with closures. In *Objects, Models, Components, Patterns*. Springer, 253–270.

G. Fraser and A. Arcuri. 2011a. Evolutionary Generation of Whole Test Suites. In *International Conference On Quality Software (QSIC)*. IEEE Computer Society, 31–40.

G. Fraser and A. Arcuri. 2011b. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software.. In *ACM Symposium on the Foundations of Software Engineering (FSE)*. 416–419.

G. Fraser and A. Arcuri. 2012. Sound Empirical Evidence in Software Testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 178–188.

Gordon Fraser and Andrea Arcuri. 2013a. EvoSuite at the SBST 2013 Tool Competition. In *International Workshop on Search-Based Software Testing (SBST)*. 406–409.

G. Fraser and A. Arcuri. 2013b. EvoSuite: On The Challenges of Test Case Generation in the Real World (Tool Paper). In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Gordon Fraser and Andrea Arcuri. 2013c. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

Gordon Fraser and Andrea Arcuri. 2014. Achieving Scalable Mutation-based Generation of Whole Test Suites. *Empirical Software Engineering (EMSE)* (2014). To appear.

Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering (TSE)* 28, 2 (2012), 278–292.

J.P. Galeotti, N. Rosner, C.G. López Pombo, and M.F. Frias. 2010. Analysis of invariants for efficient bounded verification. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 25–36.

Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2013. Improving Search-based Test Suite Generation with Dynamic Symbolic Execution. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Chen Fu, Qing Xie, Sangmin Park, Kunal Taneja, and B. M. Mainul Hossain. 2012. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. In *Proceedings of the 2012 Workshop on Dynamic Analysis (WODA 2012)*. ACM, New York, NY, USA, 1–6. DOI:http://dx.doi.org/10.1145/2338966.2336798

K. Inkumsah and T. Xie. 2008. Improving Structural Testing of Object-Oriented Programs via Integrating Evolutionary Testing and Symbolic Execution. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. 297–306.

Konrad Jamrozik, Gordon Fraser, Nikolai Tillmann, and Jonathan De Halleux. 2012. Augmented dynamic symbolic execution. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 254–257. DOI:http://dx.doi.org/10.1145/2351676.2351716

Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 159–170.

Vijay Ganesh Karthick Jayaraman, David Harvison and Adam Kiezun. 2009. jFuzz: A Concolic WhiteBox Fuzzer for Java. In *Proceedings of NASA Formal Methods Workshop (NFM 2009)*.

A Gunes Koru, Dongsong Zhang, and Hongfang Liu. 2007. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 10.

Gunes Koru, Hongfang Liu, Dongsong Zhang, and Khaled El Emam. 2010. Testing the theory of relative defect proneness for closed-source software. *Empirical Software Engineering* 15, 6 (2010), 577–598.

Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report). In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 380–389.

D. Marinov and S. Khurshid. 2001. TestEra: A Novel Framework for Testing Java Programs. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*.

P. McMinn, M. Shahbaz, and M. Stevenson. 2012. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in Software Engineering Research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 466–476. DOI:http://dx.doi.org/10.1145/2491411.2491415

Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. 2012. BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 727–737.

Catherine Oriat. 2005. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *Quality of Software Architectures and Software Quality (Lecture Notes in Computer Science)*, Vol. 3712/2005. Springer Berlin, Heidelberg, 242–256. DOI:http://dx.doi.org/10.1007/11558569_18

M. Oriol. 2012. Random testing: evaluation of a law describing the number of faults found. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 201–210.

Carlos Pacheco and Michael D. Ernst. 2005. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*. 504–527.

C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. 2007. Feedback-directed random test generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 75–84.

Sangmin Park, B. M. Mainul Hossain, Ishtiaque Hussain, Christoph Csallner, Mark Grechanik, Kunal Taneja, Chen Fu, and Qing Xie. 2012. CarFast: achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 35, 11 pages. DOI:http://dx.doi.org/10.1145/2393596.2393636

Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proc. of the 2008 Int. Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 15–26.

J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. 2009. Test Case Evaluation and Input Domain Reduction Strategies for the Evolutionary Testing of Object-Oriented Software. *Information and Software Technology* 51, 11 (2009), 1534–1548.

J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. 2010. Enabling Object Reuse on Genetic Programming-based Approaches to Object-Oriented Evolutionary Testing. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*. 220–231.

Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*. IEEE Computer Society, Washington, DC, USA, 23–32.

A. Sakti, Y.G. Guéhéneuc, and G. Pesant. 2012. Boosting Search Based Testing by Using Constraint Based Testing. *Search Based Software Engineering* (2012), 213–227.

Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification*, Thomas Ball and Robert Jones (Eds.). Lecture Notes in Computer Science, Vol. 4144. Springer Berlin / Heidelberg, 419–423.

R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. 2011. Testing Container Classes: Random or Systematic?. In *Fundamental Approaches to Software Engineering (FASE)*.

Anthony J. Simons. 2007. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engg.* 14 (December 2007), 369–418. Issue 4. DOI:http://dx.doi.org/10.1007/s10515-007-0015-3

M. Staats and C. Pasareanu. 2010. Parallel symbolic execution for structural test generation. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 183–194.

K. Taneja and Tao Xie. 2008. DiffGen: Automated Regression Unit-Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 407–410.

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. 2010. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. 336–345.

Suresh Thummalapenta, Jonathan de Halleux, Nikolai Tillmann, and Scott Wadsworth. 2010. DyGen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proc. Tests and Proofs (TAP'10)*. Springer-Verlag, Berlin, Heidelberg, 77–93.

Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: object-oriented unit-test generation via mining source code. In *Proceedings 7th European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 193–202.

Nikolai Tillmann and Wolfram Schulte. 2005. Parameterized unit tests. In *Proc. of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 253–262.

P. Tonella. 2004. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 119–128.

Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test input generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 97–107.

Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. 2006. Test input generation for Java containers using state matching. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis (ISSTA '06)*. ACM, New York, NY, USA, 37–48.

S. Wappler and J. Wegener. 2006. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference (GECCO)*. 1925–1932.

E.J. Weyuker. 1988. Evaluating software complexity measures. *IEEE Transactions on Software Engineering (TSE)* 14, 9 (1988), 1357–1365.

Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. Precise identification of problems for structural test generation. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 611–620.

T. Xie, D. Marinov, and D. Notkin. 2004. Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. 196–205.

T. Xie, D. Marinov, W. Schulte, and D. Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *Proceedings of the 11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 365–381.

Tao Xie and David Notkin. 2006. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engg.* 13 (July 2006), 345–371. Issue 3. DOI: http://dx.doi.org/10.1007/s10851-006-8530-6

Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. 2010. Test generation via Dynamic Symbolic Execution for mutation testing. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, 1–10.

Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. 2011. Combined Static and Dynamic Automated Test Generation. In *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Toronto, Canada.