Sound Empirical Evidence in Software Testing

Gordon Fraser
Saarland University
Saarbrücken, Germany
Email: fraser@cs.uni-saarland.de

Andrea Arcuri
Certus Software V&V Center, Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
Email: arcuri@simula.no

***Abstract*—Several promising techniques have been proposed to automate different tasks in software testing, such as test data generation for object-oriented software. However, reported studies in the literature only show the feasibility of the proposed techniques, because the choice of the employed artifacts in the case studies (e.g., software applications) is usually done in a non-systematic way. The chosen case study might be biased, and so it might not be a valid representative of the addressed type of software (e.g., internet applications and embedded systems). The common trend seems to be to accept this fact and get over it by simply discussing it in a threats to validity section. In this paper, we evaluate search-based software testing (in particular the EvoSuite tool) when applied to test data generation for open source projects. To achieve sound empirical results, we randomly selected 100 Java projects from SourceForge, which is the most popular open source repository (more than 300,000 projects with more than two million registered users). The resulting case study not only is very large (8784 public classes for a total of 291,639 bytecode level branches), but more importantly it is statistically sound and representative for open source projects. Results show that while high coverage on commonly used types of classes is achievable, in practice environmental dependencies prohibit such high coverage, which clearly points out essential future research directions. To support this future research, our SF100 case study can serve as a much needed benchmark for test generation.**

***Keywords*-test case generation; unit testing; search-based software engineering; benchmark**

I. Introduction

Software testing is an essential yet expensive activity in software development, therefore much research effort has been put into the question of how to automate it as much as possible. In this paper, we focus on test data generation for code coverage, in particular branch coverage in the context of object-oriented software. The simplest automated testing technique in this context is perhaps *random testing* [17], but during the years different sophisticated techniques have also been proposed. At a high level, the current state of the art can roughly be divided into three main groups: variants of random testing (e.g., Randoop [31]), *dynamic symbolic execution* (e.g., CUTE [36]) and *search-based software testing* (e.g., [28]). A recent trend also goes towards combining the individual techniques (e.g., [26]).

For "simple" techniques such as random testing, it is possible to provide rigorous answers based on theoretical analyses (e.g., see [6]). 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, among which there is the choice of the case study. If a technique works well in the lab on a specific case study, will it also work well in the real-world when it is applied by practitioners on their software? It might be that a novel technique works well in the lab just because the case study is too simple or small, but then it might fail on real-world instances. Even if real-world instances are used in a case study, the proposed technique might be too specific/biased toward those instances, and still fail when applied on new instances by practitioners.

How are case studies chosen in the literature? In most cases, this choice is not made in a systematic way, i.e., researchers choose software artifacts without providing any specific and unbiased motivation. Notice that, in many software testing contexts, this is the only viable option. This is a typical example in the context of testing techniques targeted for industrial systems. Obtaining real data from industry is a very difficult and time consuming activity, and so case studies tend to be either "small" or biased toward a specific kind of software (e.g., software in the automotive industry [48], seismic acquisition systems [5], video-conference and safety-critical control systems [22]).

The case of test data generation for *open source* software is very different from industrial software. The world wide web hosts a huge amount of open source projects, and there are specialized repositories that are freely accessible (e.g., SourceForge[1] or Google Code[2]). A researcher can easily download open source software and use those programs as case study. But how to choose them? For example, it is quite common that empirical studies *only* involve container classes (e.g., lists and vectors, see [4], [45]). It is quite hard to generalize the conclusions from such empirical studies to any other kind of software. Even when case studies are *large* and *variegated* (e.g., several hundreds of classes from different kinds of software [18], [31]), still a manual choice of software artifacts might introduce bias in the results. For

[1]http://sourceforge.net/, accessed September 2011.
[2]http://code.google.com/, accessed September 2011.

example, if a proposed testing technique does not support file system I/O, then that kind of software might have been excluded from the case study, although programs with I/O may be very common in practice.

To the best of our knowledge, we are not aware of any empirical study in the literature in which this kind of *threats to external validity* has been addressed. To cope with this problem, in this paper we present what is perhaps the first empirical study where the choice of the case study is statistically sound, as far as open source software is concerned. We randomly selected 100 Java projects from SourceForge, which is the most popular open source repository. Currently, it hosts more than 300,000 projects in several programming languages and it has more than two million registered users. The resulting case study is very large, consisting of 8784 classes for a total of 291,639 bytecode level branches. Because the case study 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 (a more precise definition will be presented later in the paper).

On this large case study we applied EvoSuite [18], [19], which is a search-based test data generation tool for object-oriented software written in Java. EvoSuite is an advanced research prototype that can efficiently handle all the different kinds of programming structures in Java (e.g., it has specific operators to handle string objects and arrays). Furthermore, it 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 feature was essential for the chosen case study, as 100 real-world open source programs likely have at least one unsafe operation.

The results of our empirical analysis show that, as demonstrated by previous empirical studies, test generation can indeed achieve high coverage – but only on a certain type of classes. In practice, dependencies on the environment inhibit high coverage, and thus clearly point out directions into which future research needs to investigate more.

In many research disciplines, common benchmarks allow tool comparisons and exploration of novel ideas – in the field of software testing there is no such common benchmark, despite recent community efforts to provide one. Our selection of 100 SourceForge projects (which we provide to the research community) can serve as a benchmark for the field of test generation for object-oriented software. We call this benchmark SF100.

The paper is organized as follows. Section II surveys the literature on test generation for object-oriented software to gain insights into the current practice in performing experiments. Section III then describes the first sound experiment in software testing, which allows us to draw conclusions about where the actual problems in this domain are. Based on these results, Section IV discusses the threats of choosing an unsuitable case study, and Section V concludes the paper.

Table I
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 case studies were chosen from available open source projects (OS), industry projects, taken from the literature, or created by the authors.

Tool	Reference	Projects	Classes	Container	Source
Artoo	[11]	1	8	8	Open Source
AutoTest	[12]	1	27	17	Open Source
Check'n'Crash	[14]	2	?	1	OS / Literature
Covana	[50]	2	388	-	Open Source
DiffGen	[40]	1	21	8	Literature
DSDCrasher	[15]	2	24	-	Open Source
DyGen	[41]	10	5,757	-	Industrial
Eclat	[30]	7	631	16	OS/Lit./Constr.
eCrash	[34]	1	2	2	Open Source
eCrash	[33]	1	2	2	Open Source
eToc	[44]	1	6	6	Open Source
EvaCon	[23]	1	6	6	Open Source
EvoSuite	[18]	6	727	-	OS + Industrial
Jartege	[29]	1	1	-	Constructed
JAUT	[10]	3	7	-	Constructed
JCrasher	[13]	1	8	2	Literature
JCute	[35]	1	6	6	Open Source
jFuzz	[25]	1	?	-	Open Source
JPF	[46]	1	1	1	Open Source
JPF	[47]	1	4	4	Constructed
JTest+Daikon	[53]	1	9	9	Constructed / Lit.
JWalk	[38]	6	13	-	Constructed
Korat	[9]	1	6	6	Literature
MSeqGen	[42]	2	450	-	Open Source
MuTest	[20]	10	952	-	Open Source
NightHawk	[3]	2	20	20	Literature
NightHawk	[4]	1	34	34	Open Source
OCAT	[24]	3	529	-	Open Source
Palus	[55]	6	4,664	-	OS + Industrial
Pex	[43]	2	8	-	Constructed
PexMutator	[54]	1	5	1	Open Source
Randoop	[31]	14	4,576	-	OS / Industrial
Rostra	[51]	1	11	9	Constructed / Lit.
RuteJ	[2]	1	1	1	Open Source
Symclat	[16]	5	16	12	Constructed / Lit.
Symstra	[52]	1	7	7	Literature
Symbolic JPF	[32]	1	1	-	Industrial
Symbolic JPF	[39]	6	6	4	Industrial/OS
TACO	[21]	6	6	6	OS/Lit.
Testera	[27]	4	4	2	Open Source
TestFul	[8]	4	15	12	OS + Literature
N/A	[7]	1	7	7	Open Source
N/A	[49]	2	4	4	Open Source
N/A	[1]	2	2	1	Open Source

II. Software Engineering Experimentation

To get a better picture of the current practice in evaluations in software engineering research, we surveyed the literature on test generation for object-oriented software. This is not meant to be an exhaustive and systematic survey, but rather a representative sample of the literature to motivate the work presented in this paper. Table I 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. This is of interest as container classes represent a particular type of classes that avoids many problems such as environment interaction, and recent studies have shown that even "simple" random testing can achieve high coverage on such classes [37]. Interestingly, 17 papers exclusively focus on container classes, and many other papers include container classes.

We also list how the evaluation classes were selected; interestingly, not a single paper out of those considered justifies why this particular set of classes was selected, and how this selection was done. In principle, this could mean that the presented set represents the entire set of classes on which the particular tool was ever tried on, but it could also mean that it is a subset on which the tool performs particularly well. An exception is industrial code, where often there is no choice, because the case study is selected by an industrial partner.

Out of 44 evaluations we considered in our literature survey, 29 selected their case study programs 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 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.

Xiao et al. [50] evaluated problems in structural test generation, concluding that the main problems in structural testing are related to object creation and external method calls. In related work, Jaygarl et al. [24] performed an experiment on open source libraries to determine the main reasons why branches were not covered by random testing. In their experiment, the main reason was also the problem of generating complex objects, followed by string comparisons and container object access. Out of the analyzed branches, only 3.1% were not covered because of environmental dependencies that were not satisfied. However, the results that we will present later in this paper lead to different conclusions.

III. A Statistically Sound Experiment

Section II illustrated that the choice of case studies in software engineering experiments is often unclear, resulting in a threat to the external validity of these experiments. In this section, we describe a *sound* experiment on software testing which does not suffer from this threat to external validity. Given these data, we perform a reality check on the research field of test generation for object-oriented software: How good is the state of the art really, and what are the real problems?

A. Objectives

The performance of test generation tools is commonly evaluated in terms of the achieved code 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 easily 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 branch coverage.

In Section II we saw that many case studies focus on container classes, which are often chosen simply because they are "nice" to test: There is no I/O, no interaction with the environment, no multi-threading, etc. In practice, one often uses existing libraries of container classes but wants to apply testing tools to other types of classes, which may very well try to interact with their environment. Test generation for such code is *unsafe* as the tested code might interact with its environment in undesired ways, for example by creating or deleting files. To evaluate to what extent this is the case, we want to find out how many unsafe operations are attempted during test generation. This results in the following two research questions:

RQ1:	What is the probability distribution of achievable branch coverage on open source software?
RQ2:	How often can classes execute unsafe operations?

B. The EvoSuite Tool for Search-based Test Generation

As context of our experiments we chose the EvoSuite [18], [19] tool, which automatically generates test suites for Java classes, targeting branch coverage. 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.

Chromosomes in EvoSuite are test suites, and each test suite consists of a variable number of test cases, which are sequences of method calls. Crossover produces offspring test suites by exchanging test cases from two parent individuals, and mutation either adds new randomly generated test cases, or mutates individual test cases. Mutation of test cases may add, remove, or change the method calls in a sequence. Fitness is calculated with respect to branch coverage, using a calculation based on the well-established branch distance measurement [28]. The branch distance estimates how close a branch is to evaluating to true or false for a particular run. For each branch we consider the minimum branch distance over all test cases of a test suite. The overall fitness of a test suite is the sum of these minimal values, such that an individual with 100% branch coverage has fitness 0.

Through its use of method sequences, EvoSuite can handle any datatype, and can be applied out of the box to any Java program. It only requires the bytecode to produce test suites, which it outputs in JUnit format.

Calculating the fitness value requires executing code, and if this code interacts with its environment then 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.

When running test generation on unknown code, using a sandbox in which permissions are restricted is essential. We therefore enabled the custom security manager for all our experiments. With respect to RQ2, we are interested in finding 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. However, no permissions were granted, except for three permissions which we determined necessary to run most code in the first place in our earlier experiments [18]: (1) Reading from properties, (2) Loading classes, and (3) Reflection. Except for these permissions, all other permissions were denied. This might be overly strict, and indeed finding a suitable set of permissions for test generation is a future research question.

In our previous experiments [18], we applied EvoSuite with a timeout of 10 minutes per class. As we apply the technique to a larger set of classes in this experiment, 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, after which the search always ended, except if 100% coverage was already achieved earlier. For all other settings, we used EvoSuite with its default parameter settings.

C. Case Study Selection

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 is the dominant site of this type, having more than 300,000 registered projects at the time of our experiments.

Table II
DETAILS OF THE SF100 CASE STUDY. FOR EACH PROJECT, WE REPORT HOW MANY CLASSES IT IS COMPOSED OF, AND THE TOTAL NUMBER OF BYTECODE BRANCHES.

Name	# Classes	# Branches	Name	# Classes	# Branches
ifx-framework	2189	93307	mygrid	35	1266
jcvi-javacommon	565	7347	jigen	35	631
caloriecount	524	12064	shop	32	1035
openjms	486	11744	dsachat	31	951
summa	428	13711	jaw-br	29	811
lilith	311	17063	gangup	29	991
corina	310	10731	inspirento	26	571
heal	186	6070	rif	25	488
at-robots2-j	174	2201	ext4j	23	525
lhamacaw	168	4973	fixsuite	22	519
xbus	168	4422	xisemele	21	343
jiggler	140	6325	biblestudy	21	630
dom4j	136	5702	imsmart	21	183
jnfe	128	2428	jgaap	19	222
hft-bomberman	125	1956	templateit	19	692
jiprof	101	5222	javaviewcontrol	18	3071
wheelwebtool	100	7246	tullibee	17	1185
sbmlreader2	85	4841	httpanalyzer	17	499
jdbacl	84	5188	asphodel	16	137
db-everywhere	84	1786	noen	16	138
quickserver	78	3648	diebierse	15	352
beanbin	75	986	cards24	14	323
echodep	74	3606	gsftp	14	614
jsecurity	72	998	jni-inchi	12	178
objectexplorer	70	1516	io-project	12	129
jhandballmoves	68	1507	fps370	12	325
schemaspy	67	3493	battlecry	11	705
twfbplayer	61	1178	celwars2009	11	964
nutzenportfolio	59	1835	ipcalculator	10	644
openhre	58	1468	sugar	9	135
apbsmem	52	1641	dvd-homevideo	9	332
geo-google	52	1344	bpmail	8	108
petsoar	52	523	byuic	8	703
lotus	52	228	jclo	8	143
follow	52	814	omjstate	8	80
jwbf	50	1371	saxpath	8	1064
lagoon	49	1140	sfmis	8	90
gfarcegestionfa	46	797	falselight	8	40
a4j	45	952	diffi	8	130
dash-framework	45	425	nekomud	7	57
javathena	44	2412	biff	6	825
lavalamp	43	306	classviewer	6	524
jtailgui	42	430	gae-app-manager	6	88
javabullboard	42	2197	resources4j	6	381
fim1	41	1194	dcparseargs	6	100
water-simulator	41	1074	trans-locator	5	74
jopenchart	38	693	shp2kml	4	51
newzgrabber	37	1354	jipa	2	34
feudalismgame	36	1454	templatedetails	2	125
jmca	35	2521	greencow	1	1

We based our selection on the dataset of all projects tagged as being written in the Java programming language. In total there were $48{,}109$ such projects at the time of our experiments, 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 project although in fact it was written in a different programming language. Finally, we did not succeed in compiling all of the projects, sometimes because they were too old and relying on particular Java APIs that are no longer available. Where available, we downloaded binary releases for projects we could not build, as EvoSuite does not actually require the source code for test generation. In total, we therefore had to consider 321 projects until we had a set of 100 projects in binary format.[3]

We call this case study SF100 benchmark. Table II shows the number of classes and branches per each of the 100 projects, whereas Table III presents the summarized statistics (e.g, mean and standard deviation). 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 private classes. In total, there are 8784 classes and 291,639 bytecode branches reported by EVOSUITE in this case study. 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 2189, and the number of branches in a class ranges from 0 to 2480. Furthermore, these distributions present infrequent extreme deviations, which is represented by high kurtosis values (kurtosis is the fourth moment of a distribution), and are highly skewed (skewness is the third moment of a distribution, and represents its asymmetry between its left and right probability tails). Notice that, in the *normal distribution*, skewness and kurtosis are equal to zero regardless of the variance.

Tables II and III report data only for the classes for which EVOSUITE run without problems. However, there were a further 87 classes in these projects for which EVOSUITE "crashed" without outputting any result. The reasons behind these crashes are still under investigation. At any rate, because these special cases represent only a tiny fraction of the case study, i.e. $87/8871 < 1\%$ of the case study, they do not pose any particularly serious threat to the validity of this study.

D. Results

To account for the randomness of the evolutionary search, we applied EVOSUITE to each of the selected case study objects 10 times with different random seeds and then averaged the values. In each run, we left EVOSUITE running up to two minutes. In total, running these experiments took up to $(8784 \times 10 \times 2)/(60 \times 24) = 122$ days (recall that, when 100% coverage was achieved, we stopped the search). Figure 1 shows the distribution of the coverage results per project.

EVOSUITE produces test suites per class, and each project might have some more difficult classes and some easier classes. Figure 2 therefore illustrates the distribution of coverage across the classes (RQ1). This shows that there is a large number of classes which can easily be fully covered by EVOSUITE (coverage 90%-100%), and also a large number of classes with problems (coverage 0%-10%), while the rest is evenly distributed across the 10%-90% range.

[3]The details of this selection process and the case study are available online at http://www.st.cs.uni-saarland.de/~fraser/SF100

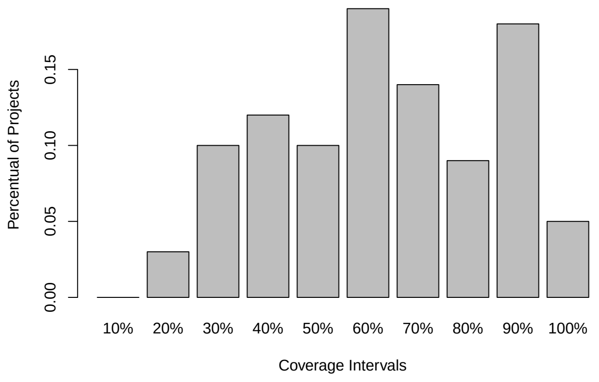

Figure 1. For each 10% code coverage interval, we report the proportion of projects that have an average coverage (averaged out of 10 runs on all their classes) within that interval. Labels show the upper limit (inclusive). For example, the group 40% represent all the projects with average coverage greater than 30% and lower or equal to 40%.

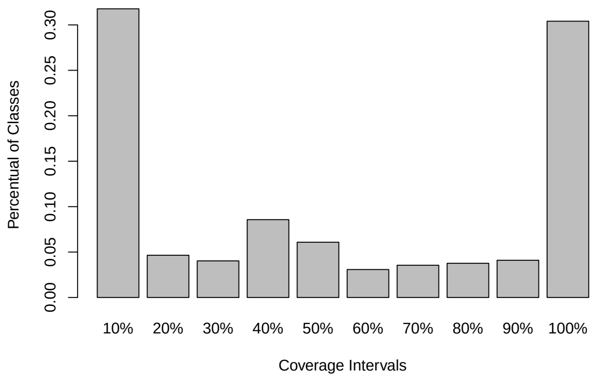

Figure 2. For each 10% code coverage interval, we report the proportion of classes that have an average coverage (averaged out of 10 runs) within that interval. Labels show the upper limit (inclusive). For example, the group 40% represent all the classes with average coverage greater than 30% and lower or equal to 40%.

The large number of classes with full coverage suggests that there are many classes that are trivially covered by EVOSUITE. To analyze this further, Figure 3 illustrates, for each 10% code coverage interval, the average number of branches 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.

On the other hand, the large number of classes that apparently have problems (0%-10% coverage) is very large. A possible reason for low 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 IV lists the average coverage achieved

Table III
SUMMARIZED STATISTICS OF THE SF100 THE CASE STUDY.

	Min	Median	Average	Std. Deviation	Max	Skewness	Kurtosis	Total
# of Classes per Project	1	35	87.84	237.00	2189	7.30	63.46	8784
# of Branches per Class	0	18	33.20	75.79	2480	16.66	429.14	291639

Table IV
FOR EACH TYPE OF PERMISSION EXCEPTION, WE REPORT IN HOW MANY CLASSES IT IS THROWN AT LEAST ONCE, AND THE AVERAGE COVERAGE FOR THOSE CLASSES. WE ALSO SHOW HOW MANY PROJECTS HAVE AT LEAST ONE CLASS IN WHICH SUCH EXCEPTION IS THROWN, AND THE AVERAGE COVERAGE FOR THOSE PROJECTS (INCLUDING ALSO THE CLASSES IN THOSE PROJECTS FOR WHICH THAT KIND OF EXCEPTION IS THROWN).

Type	Per Class		Per Project	
	Occurrence	Mean Coverage	Occurence	Mean Coverage
No Exception	0.093	0.90	0.03	0.91
AllPermission	0.00	-	0.00	-
SecurityPermission	0.11	0.54	0.36	0.51
UnresolvedPermission	0.00	-	0.00	-
AWTPermission	0.00	-	0.00	-
FilePermission	0.71	0.41	0.87	0.54
SerializablePermission	2e-04	0.79	0.01	0.44
ReflectPermission	0.00	-	0.00	-
RuntimePermission	0.52	0.49	0.85	0.55
NetPermission	0.49	0.51	0.79	0.56
SocketPermission	0.061	0.39	0.22	0.56
SQLPermission	0.00	-	0.00	-
PropertyPermission	0.074	0.50	0.16	0.59
LoggingPermission	0.00	-	0.00	-
SSLPermission	0.00	-	0.00	-
AuthPermission	0.00012	0.20	0.01	0.25
AudioPermission	0.00	-	0.00	-
OtherPermission	0.00022	0.73	0.01	0.25

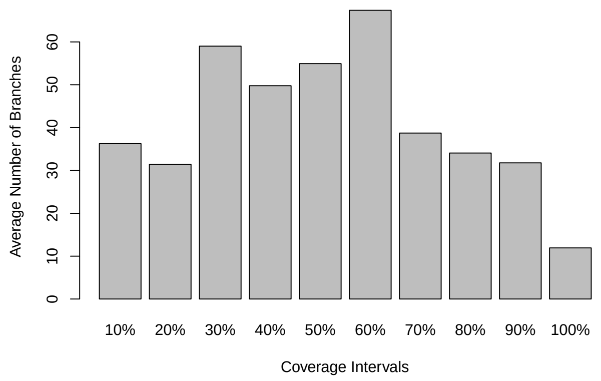

Figure 3. Average number of branches of classes within each 10% code coverage interval. Classes in the 90%-100% coverage range are the smallest, and thus potentially "easiest" classes.

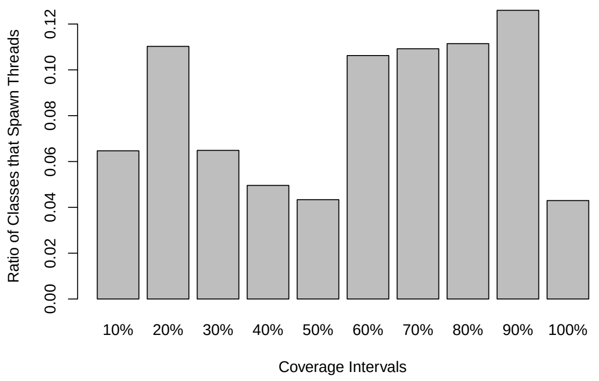

Figure 4. Average number of threads for classes within each 10% code coverage interval: Multi-threaded code does not per se inhibit coverage.

for classes for each of the possible permissions that the security manager may deny. Classes that raise no exceptions achieve an average coverage of 90%, whereas all classes that require some permission that is not granted have lower coverage. Consequently, interactions with the environment are a prime source of problems in achieving coverage (RQ2). It is striking that 71% of all classes lead to some kind of FilePermission – in other words, almost three quarters of all classes tried to access the filesystem in some way! It is important to note that this I/O might not come directly from the class under test 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 even classes that achieve high coverage often lead to some kind of denied permission check. However, the fact that classes with file access achieved significantly lower average coverage (41%) is a clear indication that file access *is* a real problem.

The other two dominant types of permissions we observed were RuntimePermissions and NetPermissions. RuntimePermissions can have various reasons, such as for example attempts to shut down the virtual machine or to access environment variables. On closer inspection, many cases of RuntimePermissions turned out to be attempts to load GUI toolkit libraries, which are not Java bytecode libraries but platform-dependent libraries. Therefore, a large part of the classes causing RuntimePermission checks (52% of the classes) are classes related to GUIs. The number of classes causing NetPermission checks is also surprisingly large (49%). Again, a NetPermission check does not automatically mean that the code under test immediately tries to access the network, but it might happen through the parameter generation sequences, and NetPermission checks are also caused for example by generation of an invalid URL. However, the Java language is by construction well-suited for web applications and several of the 100 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 threads each time any permission check was performed, and each time a test execution timed out (EvoSuite by default uses a timeout of 5s for execution of one test case). Figure 4 illustrates the relation of code coverage to the frequency of cases where we observed more than one thread: Classes that achieved 90%-100% coverage had the fewest cases of additional threads, but in general the existence of threads does not per se seem to have a big impact on coverage, as the largest number of multi-threaded classes was observed in the 80%-90% range. However, 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 (deadlocks etc), 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. However, on average we observed problems with multi-threading in only 6.4% of all projects.

E. Manual Inspection

On a high level view, the results of the experiments give us a clear message: Test generation works well as long as the environment is not involved – but usually it is involved. To understand the problems in test generation better, we manually inspected 10 classes that had low coverage but no permission problems, 10 classes that had file permission problems, 10 classes that had network problems, and 10 classes with runtime permission problems. For this selection, we sorted the classes by coverage, and then chose the classes with the lowest coverage for each of the categories, but only one per project per category.

1) Classes without permission problems: Classes with low coverage but no permission problems are of particular interest with respect to improving EvoSuite, but might not generalize to other tools. For example, in the 10 classes we investigated we identified the following main reasons for low coverage: 1) Complex string handling, 2) Java generics and dynamic type handling, and 3) branches in exception handling code.

EvoSuite has basic support for string handling; for example, it replaces calls to `String.equals` with a custom method that calculates the Levenshtein distance, which can then serve in branches to give better guidance to the search. However, this by itself is not sufficient to properly exercise complex parsers and string handling functions – at least in the two minute limit given for test generation in our experiments. However, there are dedicated string solvers and techniques to handle regular expressions, so these classes might not be problematic for other tools.

The second problem is largely due to Java's handling of generics – all type information is erased during compilation. For example, for the constructor `StateMachine(List <Transition> transitions)` EvoSuite only sees the parameter of type `List`, but not that this is supposed to be a list of `Transition` objects. When generating `List` objects EvoSuite only sees that it can add instances of type `Object` to these lists, and thus the chances of putting `Transition` objects into the list are small. These problems could be overcome by incorporating static analysis or support for type constraints. Finally, EvoSuite usually has no guidance in reaching exception handling blocks, unless there is an explicit branch in the target class that leads to a `throw` statement. Consequently, EvoSuite only covers such statements by chance.

Note that other tools might have other problems. For example, tools based on dynamic symbolic execution have more problems related to object creation [24], [50].

2) Classes with file permission checks: File handling is very common in Java classes, both in reading as well as in writing mode. Branches do not necessarily depend on file contents, but sometimes just depend on file existence or file names. However, even though these example branches do not depend on the file content, usually such branches are followed by code that manipulates these files.

Another file permission we frequently observed is when code tries to read custom property files. Even though granting read access to property files might not pose an immediate danger, such files still need to exist and contain appropriate content in order to allow testing.

Consequently, automatically setting up a suitable file environment for testing classes is a major technical obstacle.

Besides the difficulty in covering branches, there is also always the danger that code manipulating the filesystem can cause unwanted effects; for example, whenever new files are generated it is highly undesirable to let the genetic algorithm pass random strings as filenames, as that way the filesystem will be cluttered with files with random names – which is something we observed for several classes in the SF100 benchmark when deactivating the custom security manager.

3) Classes with runtime permission checks: As indicated in the previous section, a large share of the runtime permission checks we observed were due to code trying to set up a graphical user interface. To do so, Java first tries to read the environment variable `DISPLAY`, and then attempts to read a custom GUI toolkit (e.g., `jre/lib/amd64/xawt/libmawt.so`). Furthermore, most GUI applications try to access files (e.g., `.accessibility.properties`). Java has its own class of `AWTPermission` that are related to GUI events; as loading of GUI toolkits was prohibited, we did not observe any such permission checks.

We tried to see what happens when granting permissions to load libraries. However, even with these permissions the coverage does not increase, as it opens up a range of other permissions that GUI programs require: `AWTPermissions` to access the mouse pointer, a large amount of thread manipulation, special exception handlers, permissions to open windows, etc.

Besides GUI related runtime checks, there are other common permissions that are undesirable during test generation, most prominent probably the permission `exitVM.0` which is required to shut down the running virtual machine. Other instances of runtime permission checks include actions on running threads (modifyThread, stopThread), loading of libraries, queuing of printer jobs, or changing the security manager – none of these actions are desirable during test execution.

4) Classes with network permission checks: Only few classes directly attempted to open sockets (`SocketPermission`), although dependent classes or parameters did this more frequently (in total for 6% of all classes). `NetPermission`s were more frequent, and the most common type of such network permission that we observed was due to invalid URL generation (`specifyStreamHandler`). This particular permission does not immediately signify network access, but creation of a URL for a resource to which the program would normally not have access to (like file:/foo/fum/). It will require further experimentation to determine how many of these permissions were caused by the test generation itself (e.g., random strings propagating to URL generation), and how many were real attempts to access resources through URLs. In general, our observations suggest that in many cases the `NetPermission` checks are in fact very similar to `FilePermission` checks, which would mean that I/O remains the most important issue.

In general, the question of finding a perfect setting of permissions for test execution is a research question on its own, and it might be possible to increase coverage by being more gratuitous with permissions for tests.

F. Threats to Validity

Threats to *internal validity* come from how experiments were carried out. We used the EvoSuite tool for our experiments, 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. Furthermore, because EvoSuite is based on randomized algorithms, we repeated each experiment on each class 10 times to take this randomness into account. However, because our study was focused on obtaining insights on the challenges of applying test generation tools in realistic settings, our research questions did not deal with comparisons of algorithms, and so statistical tests were not required.

The main goal of this paper was to deal with the *threats to external validity* that afflict current research in software testing. The SF100 benchmark is a statistically sound representative of open source projects, and our results are also statistically valid for the 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.

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. At any rate, considering the two million subscribers of SourceForge, even if our results would be valid only for SourceForge projects, still they would be of practical value and important for a large number of practitioners (both developers and final users).

IV. Implications for Software Engineering Experimentation

In the previous section we described and analyzed a sound empirical study in software testing. Given the insights from this experiment, we now discuss the potential implications of the choice of case studies. In other words, we can answer the following research question:

RQ3: What are the consequences of choosing a small case study in a biased way?

An analysis of the literature in test data generation has shown, in Section II, that a large portion of research body has practically ignored the issues of test data generation when the system under test interacts with its environment (e.g., file systems and networks). But our empirical analysis (Section III) has shown that 90.7% of classes may lead to interactions with their environment. When there are no interactions with the environment (i.e., in the 9.3% of cases), a research prototype such as EvoSuite can achieve an average coverage as high as 90% (see Table IV). On the other hand, when we apply EvoSuite on a statistically valid sample of open source projects, the average coverage is only 48%. Therefore, our analysis casts serious doubts about the external validity of many empirical analyses that reported successful results on only a small number of classes with no interaction with their environment (e.g., container classes are a typical example).

Does using a large and variegated case study solve this problem of external validity? The answer is unfortunately *no*. If we look at Figure 1, we can see that there are 23 projects for which EvoSuite achieves on average a coverage higher than 80%. If we wanted to boast and promote our research prototype EvoSuite, we could have carried out an empirical analysis with only those 23 projects. That would have resulted in a variegated and large empirical analysis. In other words, any case study, in which the selection of artifacts is not justified and not done in systematic way, tells very little about the actual performance of the analyzed techniques.

Our empirical analysis on the SF100 benchmark clearly pointed out which are the *real* main problems in test data generation for object-oriented software. For a successful technology transfer from academic research to industrial practice, it will be essential that the research community will solve all of these problems. Therefore, we can provide the SF100 benchmark and pose this challenge to the research community:

As a research community, can we develop novel techniques that achieve on average at least 80% of branch coverage on this SF100 *benchmark?*

V. Conclusions

Experimentation in software engineering research inherently suffers from a common threat to external validity, caused by the choice of case studies for experimentation.

In this paper, we have presented the SF100 benchmark, which is a statistically sound representative of open source projects. It is composed of 100 Java projects that were randomly selected from SourceForge, which, given that it has more than 300 thousand projects and two millions subscribers, is perhaps the most used open source project repository on the web. The SF100 benchmark consists of 8784 classes, for a total of 291,639 bytecode branches. To the best of our knowledge (see Section II), this benchmark does not only represent the largest case study in the literature of test data generation for object-oriented software to date, but most importantly it is the only one that is not negatively affected by threats to external validity. External validity is one of the main barriers for a successful transfer of research results to software development practices.

On this statistically valid benchmark, we applied our research prototype EvoSuite. EvoSuite is an advanced research prototype that uses many of the most advanced techniques from the literature on search-based software testing. Our analysis shows that the large majority of classes (i.e., 90.3%) may lead to execution of "unsafe" operations, which can potentially harm the execution environment (e.g., by deleting files at random in the file systems). On classes without unsafe operations, EvoSuite achieves on average an impressive 90% branch coverage, while on the entire SF100 benchmark it "only" achieves 48% of coverage on average. As most of the research body in the software testing literature seems to ignore these issues (e.g., how to *safely* write/read on file systems and open/close network connections without negative side effects), our empirical analysis is a valuable source of statistically valid information to understand which are the *real* problems that need to be solved by the software testing research community.

With this paper, we challenge the research community to develop novel testing techniques to achieve at least 80% of branch coverage on this SF100 benchmark, because it is a valid representative of open source projects, and our EvoSuite prototype only achieved 48% coverage on average. To help the community in this regard, we provide the SF100 benchmark. For more information on EvoSuite and the SF100 benchmark, please visit our website at:

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

Acknowledgments. This project has been funded by Deutsche Forschungsgemeinschaft (DFG), grant Ze509/5-1, and by a Google Focused Research Award on "Test Amplification". Andrea Arcuri is funded by the Norwegian Research Council. We thank Jeremias Rößler for his help in acquiring the SourceForge data, and Valentin Dallmeier and Yana Mileva for feedback on earlier versions of this paper.

REFERENCES

[1] J. H. Andrews, A. Groce, M. Weston, and R. G. Xu. Random test run length and effectiveness. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 19–28, 2008.

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

[3] J. H. Andrews, F. C. H. Li, and T. Menzies. 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, pages 144–153, New York, NY, USA, 2007. ACM.

[4] J. H. Andrews, T. Menzies, and F. C. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering (TSE)*, 37(1):80–94, 2011.

[5] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 95–110, 2010.

[6] A. Arcuri, M. Z. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 219–229, 2010.

[7] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Information Sciences*, 178(15):3075–3095, 2008.

[8] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.

[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[10] F. Charreteur and A. Gotlieb. Constraint-based test input generation for java bytecode. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 131–140, Washington, DC, USA, 2010. IEEE Computer Society.

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

[12] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 72–81, 2008.

[13] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34:1025–1050, September 2004.

[14] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 422–431, New York, NY, USA, 2005. ACM.

[15] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17:8:1–8:37, May 2008.

[16] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 59–68, 2006.

[17] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering (TSE)*, 10(4):438–444, 1984.

[18] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[19] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[20] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2011.

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

[22] H. Hemmati, A. Arcuri, and L. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 327–336, 2011.

[23] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 297–306, 2008.

[24] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 159–170, New York, NY, USA, 2010. ACM.

[25] V. G. Karthick Jayaraman, David Harvison and A. Kiezun. jfuzz: A concolic whitebox fuzzer for Java. In *Proceedings of NASA Formal Methods Workshop (NFM 2009)*, 2009.

[26] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

[27] D. Marinov and S. Khurshid. Testera: A novel framework for testing java programs. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2001.

[28] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[29] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *Quality of Software Architectures and Software Quality*, volume 3712/2005 of *Lecture Notes in Computer Science*, pages 242–256, Heidelberg, 2005. Springer Berlin.

[30] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-*

Oriented Programming, 19th European Conference, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

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

[32] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 Int. Symposium on Software Testing and Analysis*, ISSTA '08, pages 15–26, New York, NY, USA, 2008. ACM.

[33] J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Test case evaluation and input domain reduction strategies for the evolutionary testing of object-oriented software. *Information and Software Technology*, 51(11):1534–1548, 2009.

[34] J. C. B. Ribeiro, M. A. Zenha-Rela, and F. F. de Vega. Enabling object reuse on genetic programming-based approaches to object-oriented evolutionary testing. In *Proceedings of the European Conference on Genetic Programming (EuroGP)*, pages 220–231, 2010.

[35] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. Jones, editors, *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer Berlin / Heidelberg, 2006.

[36] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005.

[37] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *Fundamental Approaches to Software Engineering (FASE)*, 2011.

[38] A. J. Simons. JWalk: a tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engg.*, 14:369–418, December 2007.

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

[40] K. Taneja and T. Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 407–410, Washington, DC, USA, 2008. IEEE Computer Society.

[41] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. Dygen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Proceedings of the 4th international conference on Tests and proofs*, TAP'10, pages 77–93, Berlin, Heidelberg, 2010. Springer-Verlag.

[42] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. MSeqGen: object-oriented unit-test generation via mining source code. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202, New York, NY, USA, 2009. ACM.

[43] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.

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

[45] W. Visser, C. S. Pasareanu, and R. Pelànek. Test input generation for java containers using state matching. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 37–48, 2006.

[46] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 97–107, New York, NY, USA, 2004. ACM.

[47] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 37–48, New York, NY, USA, 2006. ACM.

[48] T. Vos, A. Baars, F. Lindlar, P. Kruse, A. Windisch, and J. Wegener. Industrial Scaled Automated Structural Testing with the Evolutionary Testing Tool. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 175–184, 2010.

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

[50] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 611–620, New York, NY, USA, 2011. ACM.

[51] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 196–205, 2004.

[52] T. Xie, D. Marinov, W. Schulte, and D. Notkin. 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*, pages 365–381, 2005.

[53] T. Xie and D. Notkin. Tool-assisted unit-test generation and selection based on operational abstractions. *Automated Software Engg.*, 13:345–371, July 2006.

[54] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei. Test generation via dynamic symbolic execution for mutation testing. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[55] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA 2011, Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, Canada, July 19–21, 2011.