Behavioral Execution Comparison: Are Tests Representative of Field Behavior?

Qianqian Wang[GT] Yuriy Brun[UM] Alessandro Orso[GT]
[GT]School of Computer Science
Georgia Institute of Technology
Atlanta, GA, USA 30332-0765
{qianqian.wang, orso}@cc.gatech.edu

[UM]College of Information and Computer Science
University of Massachusetts
Amherst, MA, USA 01003-9264
brun@cs.umass.edu

***Abstract*—Software testing is the most widely used approach for assessing and improving software quality, but it is inherently incomplete and may not be representative of how the software is used in the field. This paper addresses the questions of to what extent tests represent how real users use software, and how to measure behavioral differences between test and field executions. We study four real-world systems, one used by end-users and three used by other (client) software, and compare test suites written by the systems' developers to field executions using four models of behavior: statement coverage, method coverage, mutation score, and a temporal-invariant-based model we developed. We find that developer-written test suites fail to accurately represent field executions: the tests, on average, miss 6.2% of the statements and 7.7% of the methods exercised in the field; the behavior exercised only in the field kills an extra 8.6% of the mutants; finally, the tests miss 52.6% of the behavioral invariants that occur in the field. In addition, augmenting the in-house test suites with automatically-generated tests by a tool targeting high code coverage only marginally improves the tests' behavioral representativeness. These differences between field and test executions — and in particular the finer-grained and more sophisticated ones that we measured using our invariant-based model — can provide insight for developers and suggest a better method for measuring test suite quality.**

***Index Terms*—software testing; field data; model inference**

I. Introduction

Despite its inherent limitations, testing is the most widely used method for assessing and improving software quality. One common concern with testing is that the test cases used to exercise the software in house are often not representative, or only partially representative, of real-world software use. This limits the effectiveness of the testing process. Although this limitation is well known, there is not a broad understanding of (1) the extent to which test cases may fall short in representing real-world executions, (2) the ways in which tests and real-world executions differ, and (3) what can be done to bridge this gap in an effective and efficient way. As a step toward addressing these open questions, in this paper we measure the degree to which in-house tests use software in ways representative of how real users use the software in the field.

To that end, we studied four software systems: JetUML, Apache Commons IO, Apache Commons Lang, and Apache Log4j. JetUML has two in-house test suites that achieve a relatively high coverage, and Commons IO, Commons Lang, and Log4j each have a test suite that achieves over 75% statement coverage. Because software can be used by end-users or by other (client) software, we examined both cases. We collected end-user executions for JetUML and executions through client code for Commons IO, Commons Lang, and Log4j. Specifically, we deployed JetUML to 83 human subjects who used it to perform several modeling tasks, and we collected traces of Commons IO, Commons Lang, and Log4j being used by real-world projects selected from GitHub (see Section IV-A).

To compare the behavior of in-house tests and field executions, we used four behavioral models: two coverage-based models (statements and methods covered), a mutation-based model (killed mutants, applied only to the library benchmarks because of a limitation in the execution recording tool we used), and a temporal-invariant-based model (kTails-based invariants [6], [10]). The coverage-based models represent the state of the practice in industry today for evaluating test suite quality [1], [22], [27], [28], [30]. The mutation model is the state of the art for test suite quality evaluation, but it is used sparingly in industry [33]. Finally, we designed the invariant model to further differentiate field executions from test executions at a finer-grained level. We hypothesize that the finer-grained model is better suited for identifying behavioral differences and is thus more useful in assessing test suite quality than coverage and mutation.

The results of our behavioral comparison show that, for all four models considered, field executions are different from developer-written tests in terms of the behavior they exercise. For the four systems analyzed, on average, 6.2% of statements and 7.7% of the methods executed in the field were not executed by the tests. Moreover, mutation analysis showed that adding behavior exhibited by the field executions kills 8.6% more mutants than the behavior exhibited by the developer-written tests. Finally, we found that the invariant model identified even more sizable differences between developer-written tests and field executions: 52.6% of the invariants detected in the field were missed by the tests, on average.

We also investigated whether automated test generation could help improve the representativeness of in-house tests. To do so, we augmented the developer-written tests using EvoSuite [23], an automatic test generation tool, and analyzed whether these additional tests helped decrease the gap between behavior exercised by tests and in the field. Our results show that

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.36

EvoSuite-generated tests slightly improved the test suites in terms of coverage, but not at all in terms of killed mutants and behavioral invariants. The augmented test suites still failed to exercise the behavior exercised by the user executions.

Based on the observed differences in our results, we can conclude that not only did field executions exercise code that was not tested in-house, but also — and more interestingly — some code exercised by both tests and field executions was used in a *behaviorally different* way in the two cases. Our findings provide initial evidence supporting the intuition that even high-quality test suites, both written manually and generated automatically, are typically only partially representative of field executions.

Our findings also suggest that techniques that map behavior exercised in the field to new tests are of value. We found that while coverage-based models may be too crude to provide useful information when measuring differences between tests and field executions, a mutation-based model reveals more meaningful differences that capture input domains missed by in-house tests, and a more sophisticated behavioral model based on temporal invariants captures untested execution sequences and contexts. For instance, only temporal invariants revealed behavioral differences between a field execution that triggered a defect in one of the benchmarks (JetUML) and the in-house tests. This suggests that instead of using coverage or mutation scores as a guide for improving in-house test suites, as is common in practice, using behavioral invariants may be more effective.

The main contributions of this paper are:

- A study of the differences between field executions, in-house test suites, and test suites augmented with automatically generated tests on four real-world systems.
- A methodology and a set of models for evaluating the behavioral differences between executions, including the use of temporal invariants for detecting fine-grained differences.
- A publicly available dataset of test and field executions for four real-world systems that can be used to replicate and advance our results.

The remainder of the paper is structured as follows. Section II motivates analyzing the differences between tests and field executions. Section III describes our behavioral models. Sections IV and V describe our experimental methodology and our findings, respectively. Finally, Section VI discusses related work and Section VII summarizes our contributions.

II. In-House vs. Field Executions

The goal of in-house testing is to validate that software executions adhere to an expected behavior. But the behavior tests validate may differ from the behavior exercised in the field.

While testing is the most widely used approach for assessing and improving software quality [1], it is subject to the developers' assumptions about the users' environments and behavior. These assumptions are necessary because non-trivial software cannot be tested exhaustively, and because of the vast diversity of the environments in which the software may execute, in terms of the underlying hardware, the operating system and its configuration, co-executing software, and so on. Even if it were possible to test the software in the multitude of environments representative of the users' environments, developers cannot predict all the ways in which users will use the software. In fact, developers and independent testers often envision the software is used in a prescribed manner, and these assumptions restrict the space of possible behavior considered. Beta testing aims to use real users to better approximate field executions, but beta testing is not automated, not repeatable, and only demonstrates a small slice of real-world field executions. As a result, released software typically contains bugs [3], [38], [59], most of which are unknown to the developers and do not have tests that expose them. In fact, the existence of behavioral differences between field executions and in-house tests implies that software ships with untested behavior, and thus likely unknown bugs. Thus, the quality of the software cannot be properly assessed with existing test suites, and resources cannot be properly assigned to maintenance tasks. Our work aims to identify and analyze the differences between in-house test suites and field executions from the point of view of behavior missed by the tests. This identification and analysis can provide insight into the behavior commonly missed by tests and help bridge the gap between tests and field executions, ultimately improving the effectiveness of testing.

The magnitude of the differences between tests and field executions has not been studied in detail, and there are no well understood ways of measuring these differences. While many development organizations collect data on the way their applications are used in the field, to the best of our knowledge there is no literature on the use of such data for test suite evaluation or augmentation, and no prior study has attempted to analyze and understand the differences between tests and field executions in a systematic way. Accordingly, our goals include developing a methodology for comparing executions' behavior and applying it to tests and field executions. There are two main challenges to our work:

1. There are no established standards for comparing the behavior of sets of executions. While executions can be characterized in various ways, and significant work on behavioral model inference has tackled the problem of summarizing a set of executions (e.g., [6], [7], [10], [13], [15], [39], [40], [41], [42], [43], [46], [51], [57]), this work has not considered directly comparing sets of executions. No behavioral execution representation is perfect for accurately representing all characteristics of that execution. For example, representing executions concretely with the complete trace of all executed instructions may expose trivial differences that are immaterial to the system's behavior. At the same time, a higher-level representation, such as the set of statements covered by a set of executions (a common coverage-based metric of test suite quality) is likely to fail to distinguish between two sets of executions that cover different behavior but happen to execute the same statements in alternate orders. In other words, it is difficult to filter trivial differences without also filtering important ones.

2. Collecting field-use data is difficult. It is difficult to collect field data, as it requires a large number of users who utilize the software in a real-world setting. There are costs associated with collecting field data in terms of computational overhead and storage space that may affect the users' experience. These costs must be weighed against the benefits of collection. Furthermore, collecting data from real users poses a privacy challenge.

We address the first challenge by considering four behavioral models: statement coverage, method coverage, a mutation-based model, and a temporal invariant model of behavior. We compute differences with respect to each of these models between in-house tests and field executions and evaluate each model's effectiveness at representing behavioral differences. Section III-A describes these models.

We address the second challenge in two ways. First, we turn to the Massive Open Online Course (MOOC) medium. MOOCs have gained popularity in recent years, and this popularity has provided us the opportunity to conduct studies with professional developers taking MOOCs. By targeting MOOCs, we were able to access a relatively large user base and evaluate software already used as part of the course, providing us with realistic executions. Second, we turn to the open-source software movement to access real-world systems that use library software and exercise that library software in realistic ways representative of field executions.

III. Behavioral Models for Difference Measuring

We use four behavioral models (Section III-A) to measure the differences (Section III-B) between test and field executions.

A. Behavioral models

A behavioral model is a set of entities that describe behavior exhibited in an execution. We present four behavioral models: a set of source code statements covered by executions, a set of methods covered by executions, a set of mutants killed by executions, and a set of temporal invariants over executed methods that hold over the executions.

Coverage, a commonly-used measure of test-suite quality (e.g., [1], [22], [27], [28], [30]), measures the fraction of statements, methods, branches, paths, or other code elements touched by a set of executions. For example, statement coverage of a test suite is the fraction of the source code statements in the system under test that are executed by that test suite. Coverage can typically be computed during execution with a relatively low overhead. Our coverage models model behavior exercised by an execution as the set of executed statements and the set of executed methods in that execution.

Mutation testing [17] can be used to evaluate the quality of existing tests [33] and create new tests [24]. It generates *mutants* by systematically seeding the program with artificial faults by using a set of mutation operators. Each mutant may behave slightly differently from the original program and represents a potential error a developer may have made. If a test that passes on the program fails on a mutant, the test is said to kill the mutant. Our mutation model uses the set of killed mutants in an execution to represent the behavior of that execution. For example, mutants killed by field executions but not by tests represent a difference between field and test behavior.

Trace A: `read`, `readFirstBytes`, `getBOM`, `close`
Trace B: `read`, `readFirstBytes`, `getBOM`, `length`

kTails invariants:
`getBOM` → `length` → `END`
`getBOM` → `close` → `END`
`readFirstBytes` → `getBOM` → `length` | `close`
`close` → `END`
`read` → `readFirstBytes`
`length` → `END`
`readFirstBytes` → `getBOM`
`read` → `readFirstBytes` → `getBOM`
`getBOM` → `length` | `close`
`START` → `read`
`START` → `read` → `readFirstBytes`

Fig. 1: Two example Commons IO execution traces and the kTails invariants in those traces. → means "can be immediately followed by", and each invariant can have up to k →s ($k = 2$ in our case). | means "or", and we use it as shorthand to display multiple invariants.

Temporal Invariants. Extensive work on behavioral model inference (e.g., [6], [7], [10], [13], [15], [39], [40], [41], [42], [43], [46], [51], [57]) has identified techniques for inferring precise models that describe software behavior. Our intent is to use a representative model inference algorithm, kTails [10], as a way of modeling the behavior exhibited by a set of executions. Section VI discusses other model inference algorithms and how our approach can be extended to use them. kTails serves as the basis for many other behavioral model inference algorithms (e.g., [15], [35], [39], [40], [41], [42], [43], [51]) and has been shown to (1) infer precise models [35] and (2) scale to large sets of executions [6].

The inputs of the kTails algorithm are an integer k and a set of execution traces, each a totally-ordered set of events that take place during one execution. The algorithm represents each trace as a linear finite state machine and iteratively merges the events that are followed by the same sequence of up to length k. A recent formulation of the kTails algorithm, *InvariMint*, demonstrated that the final FSM can be uniquely described by the set of specific types of temporal invariants mined from the execution traces [6], [7]; we use this formulation in our work. That is, we use InvariMint to mine temporal kTails invariants from the field and from test execution logs and compare those sets of invariants. For our experiments, we found that using $k = 2$ worked well in practice, whereas larger k caused InvariMint to run out of memory for some long traces. Models using larger k values would capture more execution data, but we leave for future work the investigation of whether the extra data captures useful behavioral information that would otherwise be missed.

We mine invariants at the level of methods executed by a trace. For example, if during an execution method `open` executed just before method `close`, then we would mine the invariant `open` $\rightarrow$ `close`. We elected to work at the method level because prior work has argued that method call sequences represent the best cost-benefit tradeoff for reproducing field failures [31]. Fig. 1 shows two examples of simplified traces from Commons IO executions and the kTails invariants ($k = 2$) mined from them. For example, the line `readFirstBytes` $\rightarrow$ `getBOM` $\rightarrow$ `length` | `close` represents two invariants: one that says that methods `readFirstBytes`, `getBOM`, and `length` occurred consecutively in an execution; and another that says tha methods `readFirstBytes`, `getBOM`, and `close` did as well.

B. Behavioral comparison metrics

For each of the behavioral models considered, we measure the similarity ($\mathbb{S}$) and unidirectional difference ($\mathbb{D}$) between field and test executions:

$$\mathbb{S} = \frac{|covered(field) \cap covered(test)|}{|covered(field) \cup covered(test)|} \quad (1)$$

$$\mathbb{D}_{tf} = \frac{|covered(test) \setminus covered(field)|}{|covered(test)|} \quad (2)$$

$$\mathbb{D}_{ft} = \frac{|covered(field) \setminus covered(test)|}{|covered(field)|} \quad (3)$$

where $covered(field)$ is the set of entities (e.g., statements, methods, mutants, or invariants) covered by field executions and $covered(test)$ is the set of entities covered by tests.

Similarity ($\mathbb{S}$) measures the fraction of the entities that are common to the field and test executions. By contrast, difference ($\mathbb{D}$) measures the fraction of the entities present in one set of executions that are not present in the other set. For example, $\mathbb{D}_{ft}$ is the fraction of the behavior exercised in the field that is not exercised by the tests.

IV. Experiment Methodology

Our study aims to answer three research questions:

RQ1: How does in-field behavior differ from the behavior exercised by developer-written tests?

RQ2: Does augmenting developer-written tests with automatically-generated tests help make in-house test suites more representative of field behavior?

RQ3: Which of the four behavioral models considered are most effective for comparing developer-written tests and field executions?

For RQ1 and RQ2, we computed each of the four behavioral models considered (see Section III-A) for each of the four benchmarks described in Section IV-A.[1] We then computed the $\mathbb{S}$ and $\mathbb{D}$ metrics (see Section III-B). We also manually qualitatively examined the behavior identified by the models as being exercised in the field but not by the tests.

[1]We did not analyze mutation models for JetUML due to a limitation of the record-and-replay tool we used to log JetUML field executions, which prevented us from replaying executions on mutated code.

benchmark	#methods	LOC	#tests	stmt. cov.
JetUML	603	8,836	*	79.9%
unit tests			53	26.4%
system tests			97	72.9%
Commons IO	940	9,682	1,125	88.8%
Commons Lang	2,647	25,570	3,735	93.3%
Log4j	1,874	21,326	591	76.8%

Fig. 2: The four benchmarks used in our study and their developer-written tests. *JetUML has developer-written unit and system tests; their combined statement coverage (**stmt. cov.**) is 79.9%. The system tests consist of 97 steps.

For RQ2, we also augmented the developer-written test suites for our benchmarks with EvoSuite [23], an automated input generation tool for Java programs that is used frequently in software engineering research (e.g., [4], [52], [54]). For each benchmark, we ran EvoSuite five times with different seeds, its default configuration, and a 5-minute time limit per class. This process generated five test suites per benchmark, where each of the test suites took between 7 to 12 hours to generate. We then measured how adding each generated test suite to the developer-written tests affected the metrics with respect to the field executions.

For RQ3, we compared the behavioral differences between field executions and developer-written tests found by our four models. We measured how much extra behavior exercised in the field each of the four models could find. We also identified which models revealed behavioral differences between the in-house tests and a field execution that triggered a defect in one of the benchmarks (JetUML).

A. Software Benchmarks

Fig. 2 describes the four benchmarks used in our study. JetUML (`http://cs.mcgill.ca/~martin/jetuml`) is a mature editor for UML diagrams. To obtain field uses for JetUML, we had 83 human subjects use JetUML Version 0.7 to design class diagrams for their course projects. The subjects were students enrolled in Georgia Tech's Online MS in CS (OMSCS) program (`http://www.omscs.gatech.edu`) whose participants are predominantly professional developers. Our study did not introduce JetUML to the students; rather, we selected to use JetUML in part because it was already used regularly in the class prior to our study. We believe that uses of JetUML in this setting can be considered realistic field uses, as they involve real users that utilize the tool to create an actual design.

The version of JetUML we used has two test suites (this changed for later versions of JetUML): a JUnit test suite and a set of system tests, both written by JetUML's developers. The JUnit tests exercise the functionality of JetUML's underlying framework. The system tests consist of a set of steps to be executed manually. As the instructions describe a single, continuous execution, we consider this as a single system test, made up of 97 steps that exercise the major functionality of the GUI, such as node and edge creation and manipulation. Fig. 2

summarizes the size of JetUML and its two test suites. together, the two test suites achieve a relatively high coverage, with the system tests achieving a considerably higher coverage than the unit tests, so we consider JetUML well tested in-house.

Apache Commons IO, Apache Commons Lang, and Apache Log4j are software libraries that provide other programs well-defined APIs. We call *client projects* other software that uses these libraries and consider these uses field uses. The three libraries we selected are widely popular and in the top 1% of projects with the highest number of client projects [53]. Moreover, these libraries have developer-written unit test suites with statement coverage of 77% or above. Fig. 2 provides summary information on the libraries and their test suites.

Collecting field executions. JetUML and the library benchmarks required different methods for collecting field executions. For JetUML, we collected field data by recording the executions performed by real users as they used the software. To do so, we used Chronon (`http://chrononsystems.com`), a record-replay tool for Java programs. During replay, in particular, we used the Post Execution Logging plugin of Chronon to log traces of executed methods and statements. Overall, 147 human subjects submitted their recorded JetUML executions. We filtered out submissions that were missing metadata needed to replay the executions or consisted entirely of opening and immediately closing the application, leaving us with field execution data from 83 human subjects.

To collect field executions for the three library benchmarks, we selected five open-source client projects for each library. To do so, we considered the results of a search on GitHub for projects whose Maven build file (`pom.xml`) listed a dependency on that library. For example, for Log4j, we searched for "pom.xml contains ⟨artifactId⟩log4j⟨/artifactId⟩". Maven is one of the most popular Java build tools [45] and all the benchmark libraries we considered use Maven. We thus looked for client projects that build with Maven as well to simplify our data collection process. We eliminated projects that did not have a README file with instructions on how to build and use the software, did not compile with the latest version of the benchmark library, or did not have tests that exercised the library. We kept the first five projects in the order of GitHub's search results that satisfied these criteria.

We then ran the client projects' tests. We consider these tests a reasonable proxy for field uses of the libraries, as they use the library as is needed by the client project [35]. Critically, these executions are not library test suites but rather tests of the projects that use the libraries, and thus accurately represent how the libraries are used in the field.

Building models. To compute method and statement coverage for our coverage analysis, we used Cobertura (`http://cobertura.github.io/cobertura/`), a widely used code coverage tool. To record method sequences and build invariant models with InvariMint [6], [7], we built an instrumentation engine on top of ASM (`http://asm.ow2.org/`). To generate mutants, we used the Defects4J framework [32]. Defects4J mutation analysis is built on top of the MAJOR mutation framework [34]. To check if client project executions killed mutants, we ran the tests of the client projects using mutated versions of the libraries.

It is worth noting that, because JetUML is a comprehensive UML editor that supports the creation and editing of many kinds of diagrams, its tests cover behavior related to all such diagrams. Since our users were only asked to create class diagrams, in our study we selected the JetUML tests that exercised features related to class diagrams. Similarly, for the library benchmarks, most client projects only used a small part of the libraries considered. In our study, we therefore excluded the classes that none of the client projects executed and the mutants therein.

B. Computing behavioral comparisons

Fig. 3 summarizes our methodology for comparing tests and field executions, which consists of four steps:

Step 1: For each benchmark, we ran developer-written tests to generate execution traces and build four behavioral models for those tests: (1) a statement coverage model, (2) a method coverage model, (3) a mutation model (only for library benchmarks), and (4) a method-level invariant model.

Step 2: We used EvoSuite to augment three of our benchmarks' developer-written test suites (see Section IV).[2] We generated five test suites for each system; each augmented test suite consisted of all the developer-written tests and one of the five EvoSuite-generated suites. We then computed, also for these augmented test suites, the four behavioral models considered. (Recall, however, that we did not compute mutation models for JetUML because our record-replay framework did not allow us to replay field executions on mutants.)

Step 3: For each benchmark, we collected field executions and computed our four behavioral models for those executions.

Step 4: For each of the four types of models, we computed the similarity ($\mathbb{S}$) and difference ($\mathbb{D}$) metrics (see Section III-B) between field and developer-written test executions, and field and automatically-augmented test executions.

C. Publicly released dataset

Our dataset includes:

- Source code and developer-written unit and system test cases for JetUML Version 0.7.
- Recorded JetUML field executions from 83 real users. The recorded executions can be used to extract code coverage, execution traces, and program states.[3]
- Source code and test suites of Commons IO, Commons Lang, Log4j, and the client projects we used in our study.
- Test suites automatically generated by EvoSuite (five per benchmark).
- Instructions on how to instrument, configure, and run the benchmarks.

[2]The version of Log4j we used is incompatible with the version of EvoSuite that generates Java 7 test suites, which are required by the Major mutation framework.

[3]The executions can be replayed using Eclipse (`http://eclipse.org`) and Chronon Time Travelling Debugger (`http://chrononsystems.com/products/chronon-time-travelling-debugger`).

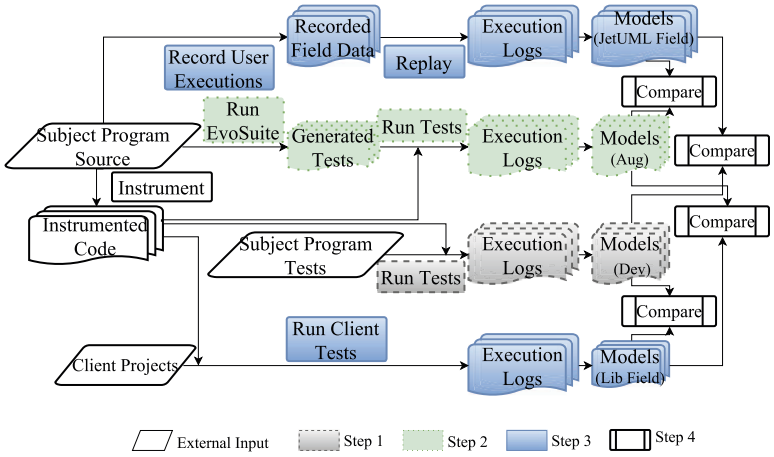

Fig. 3: Overview of the experiment setup: field data collection, model inference, and execution comparison.

Our dataset, available at `http://www.cc.gatech.edu/~orso/software/ExecutionComparison/`, can be used for analyzing the difference between test and field execution. We believe that it can also be used for studying other software engineering tasks, such as debugging and test generation.

V. Results and Discussion

RQ1: How does in-field behavior differ from the behavior exercised by developer-written tests?

Fig. 4 shows the computed $\mathbb{S}$ and $\mathbb{D}$ metrics for the four models for each of the four benchmarks.

Statement coverage. The similarity between field and developer-written test executions ranges from 18.3% (Commons Lang) to 62.1% (JetUML), with an average of 36.9%. Similarity can be low when one of the sets of executions does not exercise much of the system behavior. Because the field executions that we considered involve a relatively small subset of the user population (both in the case of actual JetUML users and in the case of client projects for the considered libraries), it is not surprising that they use only a subset of the functionality (e.g., methods) of the classes considered. This is also confirmed by the relatively high $\mathbb{D}_{tf}$ measures, which are between 31.0% (JetUML) and 81.7% (Commons Lang), with an average of 60.7%. Note that higher fractions of statements covered in testing that are not covered in the field result in lower similarity values.

The fraction of statements covered in the field that are not covered in testing, conversely, is relatively low, ranging from 0.3% (Commons Lang) to 13.0% (JetUML), with an average of 6.2%. Thus, overall, the statement coverage metric shows that field executions do differ from test executions, with 6.2% of the statements executed in the field not having been executed during testing.

For JetUML, system tests are much more similar to field executions than unit tests are (62.2% vs. 8.3%), but still 16% of the statements executed in the field are not executed by system tests. Combining unit tests and system tests reduces the similarity only by 0.1% but reduces the untested statements by 3.0%, supporting the intuition that system and unit tests complement each other. JetUML's unit tests have low coverage (recall Fig. 2) so it is not surprising that they miss statements covered in the field.

To get a better idea of the specific field behavior that in-house test may miss, we looked at the details of JetUML's statement coverage. We observed that often tests missed exceptional or corner cases. For example, they missed exception-handling code, cancellation of operations, and null inputs. Although it is not totally surprising that developers may miss special cases when testing, it is useful to see this confirmed in our results.

Method coverage. The fraction of methods exercised by both developer-written tests and field executions ranges from 22.6% (Commons Lang) to 64.9% (Log4J), with an average of 40.9%. The similarity is again low because the field executions do not use many parts of the systems considered. Again, the fraction of methods executed in testing that are not executed in the field is high, averaging 56.1%. The fraction of statements executed in the field that are not executed during in-house testing, conversely, ranges from 0.8% (Commons Lang) to 15.4% (JetUML), with an average of 7.7%, slightly higher than what we observed for statement coverage. Thus the method coverage metric also finds that field executions do differ from test executions.

As with statement coverage, for JetUML, system tests are more similar to field executions than unit tests (57.1% vs. 26.9%), but are better combined. Adding unit tests to the system tests reduces the fraction of methods executed in the field that are not executed by the tests from 21.2% to 15.4%. In addition to the relatively low coverage of the unit tests (recall Fig. 2), the unit tests also call methods directly, bypassing GUI event handling that takes place during field executions, and thus fail to exercise GUI interactions.

Also in this case, we looked at the details of JetUML's method coverage in more detail to get a better understanding of the behavioral differences between in-house tests and field

executions. In this case, we noticed that the tests missed many wrapper methods that encapsulate calls to external libraries, which may be due to the fact that developers trusted the libraries and considered the wrapper code to be too simple to test.

Mutation. The fraction of mutants killed by both developer-written tests and field executions ranges from 15.1% (Log4j) to 47.4% (Commons IO), with an average of 30.8%. The similarity results for mutation were higher than those for coverage for some projects, and lower for others. The mutation model captures at least all the statement-coverage behavioral differences, in that mutations in a statement cannot be killed by the tests if the tests do not cover that statement. However, because mutants are not evenly distributed among statements, there is no expectation of a direct relationship between coverage and mutation measures. As before, the field executions do not cover large parts of the systems: 67.4% of the mutants killed by the tests are not killed by the field executions.

The fraction of mutants killed in the field that are not killed by in-house testing ranges from 1.2% (Commons Lang) to 18.2% (Commons IO), with an average of 8.6%, which is slightly higher than what we observed for the coverage metrics. Therefore, the mutation similarity metric finds that field executions differ from test executions. And if killing mutants is representative of revealing real-world defects [33], 8.6% of the potential defects the field executions encounter would not be caught by in-house testing. This suggests a stronger notion of behavioral differences between field executions and in-house testing than the coverage metrics do.

Mutants killed in the field can survive the tests in two ways: the tests may not execute the mutated line or the tests may execute the line but not trigger anomalous behavior. The first case accounts for 83% of these mutants in our experiments. Both cases can help developers improve test suites; the first reveals an important line to cover with a new test, and the second provides a new input domain uncovered by the existing tests.

Temporal Invariants. The coverage and mutation models show average similarity between developer-written tests and field executions between 30.8% and 40.9%. They also show that 6.2% to 8.6% of the behavior exercised in the field was not exercised by the tests. The temporal invariant model shows much starker behavioral differences. The similarity in the temporal invariants mined from the executions ranges from 9.1% (Commons Lang) to 16.8% (JetUML), averaging 12.7%. Of the invariants observed in the tests, from 47.0% (JetUML) to 90.1% (Commons Lang) were not observed in the field. Meanwhile, between 24.3% (Commons Lang) and 80.5% (JetUML) of the invariants observed in the field were not observed during in-house testing. Despite the high coverage achieved by the JetUML test suite, in particular, 80.5% of the in-field behavior, as captured by behavioral invariants, did not occur during testing. Even for Commons IO and Commons Lang, for which test suites failed to cover less than 3% of the statements and methods that executed in the field, 35.4% and 24.3%, respectively, of the invariants observed in the field were not observed during testing. On average, 52.6% of the in-field invariants did not occur during testing, suggesting the test suites are very different from the field executions in terms of the temporal relationships between method executions.

We examined the behavior exercised in the field but not by the tests for JetUML, for which the difference is most pronounced. We found that such behavior occurs for three reasons: (1) field executions exercise code that is not covered by the tests; (2) users perform operations in a different order than the tests; and (3) users perform operations in a different program state than the tests. Most of the cases are of the 2nd variety, suggesting that some operation order is interchangeable, and that the developers who write tests sometimes incorrectly assume the order in which users perform operations. For example, all JetUML tests assume that every copy operation is immediately followed by a paste, whereas users sometimes performed a copy, cut, paste instead. This order of operations was never tested in-house. While such ordering may not affect functionality in some cases, when answering RQ3 we will show an example of a bug discovered in the field that the test suite missed for exactly this reason.

> **RQ2:** Does augmenting developer-written tests with automatically-generated tests help make in-house test suites more representative of field behavior?

For each of the systems considered in our study, we generated five test suites using EvoSuite (recall Section IV), augmented the developer-written tests with each of the five generated suites, and again measured the $\mathbb{S}$ and $\mathbb{D}$ metrics comparing the field executions with the augmented test suites. Fig. 5 presents these results; each cell represents the mean over the five augmented test suites for each benchmark.

The augmented test suites improves the statement coverage of the developer-written test suite for JetUML by 7.3%, for Commons IO by 4.8%, and for Commons Lang by 3.8%.

For JetUML, the augmented test suites have a slightly lower similarity with field executions with respect to the statement coverage (58.3% vs. 62.1%). The decrease is due to the generated tests covering more code that the field executions do not exercise. With respect to method coverage and invariant models, the similarity is higher (59.3% vs. 51.6% and 17.4% vs. 15.3%, respectively). The increase is caused by the fact that EvoSuite-generated tests are able to cover some field-executed methods that are missed by developer-written tests.

For Commons IO and Commons Lang, the augmented tests result in decreased similarity beween test runs and field executions with for all four models (see the last two rows of Fig. 5). Also in this case, the change is due to the generated tests covering additional code, mutants, and invariants that field executions did not exercise.

The augmented test suites capture more in-field behavior than the developer-written test suites for statement coverage, method coverage, and invariant models for all benchmarks. With respect to the statement coverage model, the augmented tests miss 10.9% and 0.398% of the in-field behavior (for JetUML and libraries, respectively), as compared to 13.0% and 0.403% for the developer-written tests. For the method

benchmark	tests	statement			method			mutation			temporal invariants		
		$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$
JetUML	unit	0.083	0.745	**0.893**	0.269	0.532	**0.612**				0.054	0.682	**0.939**
	system	0.622	0.288	**0.160**	0.571	0.326	**0.212**				0.133	0.518	**0.845**
	both	0.621	0.310	**0.130**	0.516	0.431	**0.154**				0.153	0.586	**0.805**
Commons IO		0.217	0.782	**0.005**	0.246	0.753	**0.022**	0.474	0.471	**0.182**	0.111	0.882	**0.354**
Common Lang		0.183	0.817	**0.003**	0.226	0.773	**0.008**	0.298	0.702	**0.012**	0.091	0.901	**0.243**
Log4j		0.456	0.518	**0.108**	0.649	0.287	**0.122**	0.151	0.848	**0.065**	0.136	0.799	**0.704**
average		0.369	0.607	**0.062**	0.409	0.561	**0.077**	0.308	0.674	**0.086**	0.127	0.763	**0.526**

Fig. 4: $\mathbb{S}$ and $\mathbb{D}$ metrics comparing field executions to developer-written tests, for statement and method coverage, mutation, and invariant models computed for the four benchmarks considered. (Recall that the record-and-replay framework did not allow us to replay JetUML field executions on mutants.) The differences observed in the field but not in the tests are **highlighted**. The average row uses JetUML's combined unit and system tests.

benchmark	#gen tests	augmented stmt. cov.	statement			method			mutation			temporal invariants		
			$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$	$\mathbb{S}$	$\mathbb{D}_{tf}$	$\mathbb{D}_{ft}$
JetUML	768	87.2%	0.583	0.370	**0.109**	0.593	0.355	**0.119**				0.174	0.598	**0.767**
IO	2,148	92.5%	0.167	0.827	**0.005**	0.237	0.762	**0.015**	0.430	0.501	**0.182**	0.093	0.902	**0.329**
Lang	4,769	97.1%	0.174	0.825	**0.003**	0.216	0.784	**0.000**	0.287	0.813	**0.012**	0.076	0.922	**0.224**
average	3,459	94.8%	0.171	0.826	**0.004**	0.227	0.773	**0.008**	0.359	0.657	**0.097**	0.085	0.912	**0.277**
average (developer tests)			0.200	0.799	**0.004**	0.236	0.763	**0.015**	0.386	0.587	**0.097**	0.101	0.892	**0.299**

Fig. 5: $\mathbb{S}$ and $\mathbb{D}$ metrics comparing field executions to developer-written test suites augmented with EvoSuite, for statement and method coverage, mutation, and invariant models for JetUML, Commons IO, and Commons Lang. We were unable to perform this analysis for Log4j because the version of Log4j in our study is incompatible with versions of EvoSuite that generate tests compatible with the Major mutation framework. Differences observed in the field but not in the tests are **highlighted**. For comparison, we include the average of developer-written tests for IO and Lang.

coverage model, the augmented tests miss 3.9% and 0.8% of the in-field behavior, as compared to 15.4% and 1.5% for the developer-written tests. For the mutation model, conversely, there is no change in terms of missed mutants covered in the field executions (both 9.7%). Finally, for the invariant model, the augmented tests miss 76.7% and 27.7% of the in-field behavior, as compared to 80.5% and 29.9% for the developer-written tests. Overall, while the representativeness of the tests is improved, there are still significant differences with the in-field behavior.

It is worth mentioning that the EvoSuite-generated tests cover more of the exceptional behavior than the developer-written tests. This is the main reason why the augmented test suites reduced the number of statements covered in the field but missed by the tests. However, the new tests that cover exceptional flows are mostly trivial and typically consisted of passing null values; most other control flows related to corner cases are not covered by the new tests. Also, EvoSuite-generated tests were unable to kill mutants that are missed by developer-written tests. With respect to invariants, new invariants induced by the new tests are largely due to the fact that EvoSuite checks a value after setting it whenever possible. This is the case, for instance, for the new invariant `GraphPanel.setModified` $\rightarrow$ `GraphPanel.isModified`. These invariants are likely not as useful for capturing important field behavior as those mined from developer-written tests.

Overall, coverage driven automated test generation helped only marginally in reducing the differences between field executions and in-house tests. Most importantly, it mainly helped in the trivial case of code for which no developer tests existed.

> **RQ3:** Which of the four behavioral models considered are most effective for comparing developer-written tests and field executions?

Overall, all models were able to detect some differences between test and field executions. The mutation models identified more differences than the statement and method coverage models (Fig. 4 and 5). In general, there is not a subsumption relationship between mutation and structural coverage. On the one hand, statement and method coverage models can reveal differences in behavior that mutation does not detect; for an example, consider the case of a statement that is covered in the field but not in house and is not selected for mutation. On the other hand, mutation models can reveal differences in behavior that statement and method coverage would miss; consider, for instance, the case of a statement that is mutated, is covered both in the field and in house, and only in the field is executed under a state that kills the mutant.

The invariant model is strictly more inclusive than method coverage models: if an invariant involving a method is mined, that method must have been executed. Using invariant models

to characterize differences between field and test executions cannot therefore miss any information reported by method coverage models. Statement coverage models could identify finer-grained differences than method-level invariants report, but the data in Fig. 4 suggest that this was uncommon. (We did not consider statement-level temporal invariant models, which would capture all such differences, but would also be impractically large.) In general, the invariant model finds more sophisticated differences than the coverage models, such as different orders of method executions, and the execution of the same methods under different program states.

We found some evidence that the differences between field and test executions that the invariants model finds but other models miss can be important. For a concrete example, when starting JetUML, users have to either select a type of diagram to create, or use the `File` menu to open or create a file. The invariant model found that users sometimes selected `Undo` from the `Edit` menu as the first action. This unexpected operation caused an exception that crashed the program. Neither developer-written nor EvoSuite-augmented test suites found this error, and the coverage and mutation models did not identify this difference. The JetUML developers have identified this as a real defect and have fixed it. In more general terms, our results suggest that (1) invariant models may be better than simpler models at discovering important behavioral differences between in-house and field executions and (2) these differences can reveal relevant behavior (including defects).

A. Summary findings

Here is a summary of our findings for the systems and executions we considered:

- Field executions can differ considerably, in terms of the behavior they exercise, from in-house test executions.
- Automatically generated tests can only marginally improve the representativeness of in-house tests.
- All behavioral models can find differences between in-house testing and field executions. Specifically, statement coverage models can identify corner cases missed during testing, while method coverage models can find high-level differences in the features used. Mutation models can miss some differences identified by the simpler coverage models, but they may be able to identify specific states not covered by the tests. Finally, invariant models subsume (at the method level) coverage models and can identify richer differences, such as differences in operation order and context.
- Unsurprisingly, using the state of the practice (coverage) or the state of the art (mutation) to assess the quality of a test suite falls short of precisely measuring how well the test suite represents field executions. An invariant-based model may be a better adequacy and selection criterion, but further investigation on infeasibility issues is required.

B. Recommending test cases based on field executions

If software could be deployed together with a model computed for the in-house test suite, then field behavioral differences with respect to that model could be used to recommend developers new tests that can improve the test suite's representativeness.

Coverage-based metrics can recommend the methods and statements uncovered by the tests that are executed in the field, as also suggested in previous work [49]. It is usually impossible to achieve 100% coverage during testing because of resource constraints and because complex code may make it difficult to devise test inputs that reach particular code elements. Field-based traces can help in two ways: first, by identifying which code elements need attention because they are used in the field; and second, by using inputs from the field to help cover these elements (e.g., [14], [31]).

While collecting field coverage data is practical in many situations, using mutation analysis is more complex. Theoretically, the in-field executions could use other cores or the cloud to simultaneously execute the system on mutants. This is an expensive proposition, but could identify interesting states unexplored during testing.

Finally, invariant models can also be computed in a fairly lightweight manner at runtime and can reveal sequences of operations users perform that the tests miss. Our results show that these differences can identify executions that lead to code defects but are missed by other models, suggesting that invariants may be the most useful field information to improve test suites. Developers who create tests based on assumptions about how the code is used in the field can consider in-field invariant violations as violations of (some of) these assumptions and may use that information to refine such assumptions and ultimately improve their tests.

In future work, we plan to investigate some of the above directions and identify practical and effective ways in which information from the field can be used to improve, possibly in an automated way, in-house test suites.

C. Threats to validity

Construct: We simulated library field executions by running the test suites of client projects of the libraries considered. These tests are written by the client project developers, who are the intended users of the libraries, without any involvement of the library developers. We therefore believe that our assumption that the client tests are representative field uses of the libraries is reasonable.

Internal: There are potential errors and mistakes in the process of writing infrastructure, building benchmarks, running experiments, and analyzing results. To reduce this threat, we carefully examined the a sample of our results to check their correctness.

External: We used four systems in our study, so our results may not generalize to other systems and executions. This limitation is an artifact of the complexity of collecting real-world field executions and of performing experiments in general. We mitigated this threat by collecting different kinds of systems, one GUI and three libraries. Another threat is that statement and method coverage models may not represent behavior as well as other coverage metrics. We used them nevertheless

because they are the two most common coverage metrics used in practice. We used EvoSuite as a representative automated test generation tool, in part because a recent study found it to be one of the most effective tools in that arena [55].

VI. Related Work

Our research is related to work in field data (Section VI-A) and behavioral representation (Section VI-B).

A. Using field data in software engineering

Researchers have investigated for over a decade the use of field data to aid various software engineering tasks traditionally performed in-house. The Gamma project, for example, aims to leverage field data to help software maintenance tasks, such as impact analysis and regression testing [47], [48]. For another example, Elbaum and Diep have investigated the potential benefit of using field data to improve software profiling [19], [20]. Our work is related to these techniques, as its ultimate goal is to use field data to improve in-house testing activities.

Our work is also related to that of Pavlopoulou and Young, who developed a technique for collecting field data about statements covered in the field but not in-house [49]. In fact, their approach could be used in our context. Hilbert and Redmiles [29] propose an agent-based approach for collecting field usage data and feedback that can provide developers with usage- and usability-related information [29]. This information can help detect and resolve mismatches between developers' expectations and actual software use. By contrast, our effort focuses on modeling and analyzing field executions.

Bug reports can also shed light on field executions and are one of the most commonly used field data types. For example, models for fault localization and automatic retrieval of faulty files based on bug reports can aid debugging [50]. These models are at least as effective as other automated debugging techniques. However, most human-written bug reports contain limited information, which reduces their utility in practice [58]. Meanwhile, automatically generated crash reports contain rich data that can help triage reports and locate bugs. ReBucket clusters duplicate crash reports collected in the field by grouping crash reports based on call stack similarity calculated using the Position Dependent Model [16]. Similarly, CrashLocator uses call stacks from the crash reports to generate approximate crash traces by stack expansion and uses the expanded traces to locate faulty methods [60]. Both techniques require large numbers of crash reports to be effective.

B. Behavioral representation

Code coverage has long been the standard metric for test suite quality [1], [12], [22], [26], [27], [28], and executions can be characterized using variations in coverage (e.g., via basic block vectors [56]). However, recent studies have shown that coverage may not be a great indicator of test suite effectiveness at finding faults [30], and that mutation kill scores are a better metric [33]. Still, stronger proxies for representing system behavior may be desirable, such as invariants-based descriptions of the behavior [21] or finite-state-machine-based models of the behavior [6], [7], [13], [15], [35], [39], [40], [41], [42], [43], [46], [51], [57].

There are numerous algorithms to mine temporal invariant instances [2]. For example, Javert [25] infers property specifications by composing simpler micro-patterns into larger ones, focusing on efficiency. *N*-grams can represent executions in terms of substrings of kernel-call or application-server-call sequences [44], which is a similar representation to kTails. We use InvariMint [6], [7] to mine behavioral properties, but our work is easily extendable to other property-mining algorithms, and advances in the richness of these algorithms are complementary and beneficial to our work. Another kind of property that our work does not include is structural, data-value properties that relate internal program variables, often described with variable values and can encode method pre- and post-conditions, as well as class-level property types. Our work can be extended to use such properties (e.g., mined by Daikon [21] from program executions), and again, advances in the inference of such properties are complementary and beneficial to our work. Combining structural and temporal properties is likely to increase the precision of behavioral difference measurement between test and field executions.

Finite-state-machine-based models that describe system behavior are similarly complementary to our work. The kTails algorithm [10] is the basis for numerous behavioral model-inference algorithms [15], [35], [39], [40], [41], [42], [43], [51]. Our work uses the behavioral invariants that precisely describe the models inferred by kTails, but could be adapted to use the invariants that describe the behavioral models inferred by each of these algorithms. InvariMint [6], [7] focuses on decomposing behavioral model inference algorithms into such invariants, including for the kTails [10] and the Synoptic [5], [9] algorithms. Here, we used InvariMint to infer the behavioral invariants, which facilitates expanding the work to include other kinds of invariants. However, some model-inference techniques require richer than standard FSM models, and may not be represented precisely and completely by behavioral, temporal properties. GK-Tails [43] requires EFSMs, and RPNI [13] requires Probabilistic FSMs. The Alergia algorithm [13] cannot be easily specified using InvariMint because of reliance on transition probabilities updated dynamically during the model inference procedure.

User-specified LTL formulae of desired system behavior can be combined, checked by a model checker, and used as constraints on inferring a single behavioral model [57]. By contrast, our work does not require the user, nor the developer, to know the desired system properties, instead comparing test executions to field executions. However, it is conceivable that checking for differences between user-specified properties in test and field executions may lead to further insights. Other representations of behavior are also possible, including UML sequence diagrams [61], communicating automata [8], [11], and symbolic message sequence graphs [36]. These behavioral representations are outside the scope of our work, but our analysis could be extended to these representations as well.

VII. Conclusions

This paper presents the first study whose goal is to understand, quantify, and analyze similarities and differences between in-house testing and in-field usage. To do so, we have used four models of software behavior — two based on coverage, one based on mutation analysis, and one based on temporal behavioral invariants. Our results show that, for all the four models considered, there are gaps between how developers test, or how they expect users to use their software, and how users actually use this software in the field.

Although still preliminary, we believe that our results are significant because they were obtained from analyzing field data collected from real systems used by real users or real developers (in the case of the libraries and their client projects). In addition, our results provide evidence of potential for several research direction that aim to bridge the gap between the way software is tested and the way it is used.

First, our findings can be used as a starting point for developing techniques that use the differences between in-house and field behavior to guide the generation of new test cases. One natural way to do so would be to generate test cases that (1) cover code exercised in the field but missed in-house, and (2) cover execution sequences that violate in-house invariants. While this approach may raise privacy concerns related to sharing detailed in-field execution data with developers, prior work has made progress addressing these concerns [14].

Second, research on using the temporal invariant model to assess test suite completeness and quality may provide a better metric for the purpose than statement coverage and mutation testing. Our results show that the temporal invariants we used are able to discriminate failing executions that coverage and mutation fail to discriminate and for which automated test generation could not generate revealing tests. A first step in this direction would be a systematic evaluation on real-world defect benchmarks (e.g., Defects4J [32] or ManyBugs [37]).

Third, while kTails invariants proved able to capture important behavioral differences, many other temporal invariants could perform just as well or even better. Systematically exploring the space of temporal patterns, perhaps starting with the most common ones used in defining system requirements [18], may reveal invariants better suited for (1) representing execution behavior, (2) efficient in-field computation, and (3) providing developers with useful information for writing new tests.

Finally, developing mechanisms for efficiently monitoring field executions, building behavioral models, and comparing live executions to models of tested behavior may make such data easier to collect and more helpful to developers for various tasks. Such mechanisms could for example generate better crash reports and feedback for developers. They could also be used to guide self-adaptation and automated program repair, steering live executions that violate behavioral models toward desired, better-tested behavior.

Acknowledgments

We thank the students of the online course who kindly agreed to let us collect field data about their use of JetUML for our study. This work was partially supported by the National Science Foundation under grants CCF-1453474, CCF-1564162, CCF-1320783, and CCF-1161821, and by funding from Google, IBM Research, and Microsoft Research.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[2] C. M. Antunes and A. L. Oliveira. Temporal data mining: An overview. In *Workshop on Temporal Data Mining*, San Francisco, CA, USA, 2001.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *Workshop on Eclipse Technology eXchange*, pages 35–39, San Diego, California, 2005.

[4] A. Arcuri, G. Fraser, and J. P. Galeotti. Generating TCP/UDP network data for automated unit test generation. In *10th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 155–165, Bergamo, Italy, 2015.

[5] I. Beschastnikh, J. Abrahamson, Y. Brun, and M. D. Ernst. Synoptic: Studying logged behavior with inferred models. In *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE)*, pages 448–451, Szeged, Hungary, September 2011.

[6] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *International Conference on Software Engineering (ICSE)*, pages 252–261, San Francisco, CA, USA, May 2013.

[7] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering (TSE)*, 41(4):408–428, April 2015.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *International Conference on Software Engineering (ICSE)*, pages 468–479, Hyderabad, India, June 2014.

[9] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 267–277, Szeged, Hungary, September 2011.

[10] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers (TC)*, 21(6):592–597, 1972.

[11] B. Bollig, J.-P. Katoen, C. Kern, and M. Leucker. Learning communicating automata from MSCs. *IEEE Transactions on Software Engineering (TSE)*, 36(3):390–408, 2010.

[12] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, USA, 2008.

[13] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference and Applications (ICGI)*, pages 139–152, Alicante, Spain, 1994.

[14] J. Clause and A. Orso. Camouflage: Automated anonymization of field data. In *International Conference on Software Engineering (ICSE)*, pages 21–30, Honolulu, HI, USA, 2011.

[15] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM TOSEM*, 7(3), 1998.

[16] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *International Conference on Software Engineering (ICSE)*, pages 1084–1093, Zurich, Switzerland, 2012.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978.

[18] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering (ICSE)*, 1999.

[19] S. Elbaum and M. Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Transactions on Software Engineering*, 31(4):312–327, 2005.

[20] S. Elbaum and M. Hardojo. An empirical study of profiling strategies for released software and their impact on testing activities. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 65–75, Boston, MA, USA, 2004.

[21] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, 2001.

[22] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 153–162, Lake Buena Vista, FL, USA, 1998.

[23] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *8th Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering Tool Demonstration Track (ESEC/FSE)*, pages 448–451, Szeged, Hungary, September 2011.

[24] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–158, Trento, Italy, 2010.

[25] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA, 2008.

[26] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 302–313, Lugano, Switzerland, 2013.

[27] A. Groce. Coverage rewarded: Test input generation via adaptation-based programming. In *International Conference on Automated Software Engineering (ASE)*, pages 380–383, 2011.

[28] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(2):145–160, Feb. 2008.

[29] D. M. Hilbert and D. F. Redmiles. Large-scale collection of usage data to inform design. In *IFIP TC.13 International conference on human-computer interaction*, pages 569–576. IOS Press, 2001.

[30] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering (ICSE)*, pages 435–445, Hyderabad, India, 2014.

[31] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *International Conference on Software Engineering (ICSE)*, pages 474–484, Zurich, Switzerland, 2012.

[32] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440, San Jose, CA, USA, July 2014.

[33] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *International Symposium on Foundations of Software Engineering (FSE)*, Hong Kong, China, 2014.

[34] R. Just, F. Schweiggert, and G. M. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a java compiler. In *International Conference on Automated Software Engineering (ASE)*, pages 612–615, Lawrence, KS, USA, 2011.

[35] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *International Symposium on the Foundations of Software Engineering (FSE)*, pages 178–189, Hong Kong, China, November 2014.

[36] S. Kumar, S.-C. Khoo, A. Roychoudhury, and D. Lo. Mining message sequence graphs. In *International Conference on Software Engineering (ICSE)*, pages 91–100, Honolulu, HI, USA, 2011.

[37] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.

[38] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–154, San Diego, CA, USA, 2003.

[39] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Working Conference on Reverse Engineering (WCRE)*, 2006.

[40] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 265–275, Portland, OR, USA, 2006.

[41] D. Lo and S. Maoz. Scenario-based and value-based specification mining: Better together. In *International Conference on Automated Software Engineering (ASE)*, pages 387–396, Antwerp, Belgium, 2010.

[42] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 345–354, Amsterdam, The Netherlands, 2009.

[43] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *International Conference on Software Engineering (ICSE)*, pages 501–510, Leipzig, Germany, 2008.

[44] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Workshop on New Security Paradigms (NSPW)*, pages 101–110, Ballycotton, County Cork, Ireland, 2000.

[45] V. Massol, T. O'Brien, and M. K. Loukides. *Maven: A developer's notebook*. O'Reilly, 1st ed edition, 2005.

[46] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *International Conference on Automated Software Engineering (ASE)*, pages 19–30, Västerås, Sweden, September 2014.

[47] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. In *The 9th European Software Engineering Conference and 11th Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 128–137, Helsinki, Finland, September 2003.

[48] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 65–69, Rome, Italy, 2002.

[49] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *International Conference on Software Engineering (ICSE)*, pages 277–284, 1999.

[50] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Mining Software Repositories (MSR)*, pages 43–52, Honolulu, HI, USA, 2011.

[51] S. P. Reiss and M. Renieris. Encoding program executions. In *International Conference on Software Engineering (ICSE)*, pages 221–230, Toronto, ON, Canada, 2001.

[52] J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 338–349, Baltimore, MD, USA, 2015.

[53] A. A. Sawant and A. Bacchelli. A dataset for api usage. In *Working Conference on Mining Software Repositories (MSR)*, pages 506–509, Florence, Italy, 2015.

[54] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201–211, Lincoln, NE, USA, November 2015.

[55] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *International Conference on Automated Software Engineering (ASE)*, pages 201–211, Lincoln, NE, USA, November 2015.

[56] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Operating Systems Review*, 36(5):45–57, Oct. 2002.

[57] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *International Conference on Automated Software Engineering (ASE)*, pages 248–257, L'Aquila, Italy, September 2008.

[58] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, Baltimore, MD, USA, 2015.

[59] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *International Workshop on Mining Software Repositories (MSR)*, Minneapolis, MN, USA, 2007.

[60] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–214, San Jose, CA, USA, 2014.

[61] T. Ziadi, M. A. A. da Silva, L. M. Hillah, and M. Ziane. A fully dynamic approach to the reverse engineering of UML sequence diagrams. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 107–116, Las Vegas, NV, USA, 2011.