Augmenting Test Oracles With Production Observations

Deepika Tiwari

Supervised by
Professors Benoit Baudry and Martin Monperrus

School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
KTH Royal Institute of Technology
Stockholm, Sweden, 2024

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Software and Computer Systems
SE-10044 Stockholm
Sweden

TRITA-EECS-AVL-2024:XX
ISBN XX

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framläggs till offentlig granskning för avläggande av Teknologie doktorexamen i elektroteknik fredagen den 13 december 2024 klockan 13.15 i Kollegiesalen, Brinellvägen 6, Stockholm.

Tryck: Universitetsservice US AB

Abstract

Software testing is the process of verifying that a software system behaves as it is intended to behave. Significant resources are invested in creating and maintaining strong test suites to ensure software quality. However, in-house tests seldom reflect all the scenarios that may occur as a software system executes in production environments. The literature on the automated generation of tests proposes valuable techniques that assist developers with their testing activities. Yet **the gap between tested behaviors and field behaviors remains largely overlooked.** Consequently, the behaviors relevant for end users are not reflected in the test suite, and the faults that may surface for end-users in the field may remain undetected by developer-written or automatically generated tests.

This thesis proposes a novel **framework for using production observations, made as a system executes in the field, in order to generate tests**. The generated tests include test inputs that are sourced from the field, and oracles that verify behaviors exhibited by the system in response to these inputs. We instantiate our framework in three distinct ways.

First, for a target project, we focus on methods that are inadequately tested by the developer-written test suite. At runtime, we capture objects that are associated with the invocations of these methods. The captured objects are used to generate tests that recreate the observed production state and contain oracles that specify the expected behavior. Our evaluation demonstrates that **this strategy results in improved test quality for the target project.**

With the second instantiation of our framework, we observe the invocations of target methods at runtime, as well as the invocations of methods called within the target method. Using the objects associated with these invocations, we generate tests that use mocks, stubs, and mock-based oracles. We find that **the generated oracles verify distinct aspects of the behaviors observed in the field, and also detect regressions within the system.**

Third, we adapt our framework to capture the arguments with which target methods are invoked, during the execution of the test suite and in the field. We generate a data provider using the union of captured arguments, which supplies values to a parameterized unit test that is derived from a developer-written unit test. Using this strategy, **we discover developer-written oracles that are actually generalizable to a larger input space.**

We evaluate the three instances of our proposed framework against real-world software projects exercised with field workloads. Our findings demonstrate that runtime observations can be harnessed to generate complete tests, with inputs and oracles. The generated tests are representative of real-world usage, and can augment developer-written test suites.

Keywords: Test generation, Test oracles, Production observations

Sammanfattning

Programvarutestning är processen för att verifiera att ett mjukvarusystem fungerar som det är tänkt att fungera. Betydande resurser investeras i att skapa och underhålla starka testsviter för att säkerställa mjukvarukvalitet. Interna tester återspeglar dock sällan alla scenarier som kan uppstå när ett mjukvarusystem körs i produktionsmiljöer. Litteraturen om automatiserad testgenerering föreslår värdefulla tekniker för att hjälpa utvecklare i deras testaktiviteter. Ändå förbises **gapet mellan testade beteenden och fältbeteenden till stor del.** Följaktligen återspeglas inte beteenden som är relevanta för slutanvändare i testsviten, och de fel som kan visas för slutanvändare i fältet kan förbli oupptäckta av utvecklarskrivna eller automatiskt genererade tester.

Denna avhandling föreslår ett nytt **ramverk för att använda produktionsobservationer, gjorda som ett system exekverar i fält, för att generera tester.** De genererade testen inkluderar testindata som kommer från fältet och orakel som verifierar beteenden som uppvisas av systemet som svar på dessa indata. Vi instansierar vårt ramverk på tre olika sätt.

Först, för ett målprojekt, fokuserar vi på metoder som är otillräckligt testade av den utvecklarskrivna testsviten. Vid körning fångar vi objekt som är associerade med anropen till dessa metoder. De fångade objekten används för att generera tester som återskapar det observerade produktionstillståndet och innehåller orakel som anger det förväntade beteendet. Vår utvärdering visar att **denna strategi resulterar i förbättrad testkvalitet för målprojektet.**

Med den andra instansieringen av vårt ramverk observerar vi anrop till målmetoder vid körning, såväl som anrop till metoder som anropas inom målmetoden. Med hjälp av objekten som är associerade med dessa anrop genererar vi tester som använder hånar, stubbar och skenande orakel. Vi finner att **de genererade oraklen verifierar distinkta aspekter av beteenden som observerats i fältet, och även upptäcker regressioner inom systemet.**

För det tredje anpassar vi vårt ramverk för att fånga de argument med vilka målmetoder anropas, under körning av testsviter och i fält. Vi genererar en dataleverantör med hjälp av föreningen av fångade argument, som tillhandahåller värden till ett parameteriserat enhetstest härlett från ett utvecklarskrivet enhetstest. Med den här strategin **upptäcker vi utvecklarskrivna orakel som faktiskt är generaliserbara till ett större inmatningsutrymme.**

Vi utvärderar de tre fallen av vårt föreslagna ramverk mot verkliga programvaruprojekt som utövas med fältarbetsbelastning. Våra resultat visar att körtidsobservationer kan utnyttjas för att generera kompletta tester, med ingångar och orakel. De genererade testerna är representativa för användning i verkligheten och kan utöka utvecklarskrivna testsviter.

Nyckelord: Testgenerering, Testorakel, Produktionsobservationer

for mummy and papa

List of Papers

This thesis includes the following contributions by the author, listed in chronological order.

I. D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production Monitoring to Improve Test Suites," *IEEE Transactions on Reliability*, vol. 71, no. 3, 2021: pp. 1381-1397,
doi: 10.1109/TR.2021.3101318

II. L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry "Harvesting Production GraphQL Requests to Detect Schema Faults," *in Proceedings of the 15th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2022: pp. 365-376,
doi: 10.1109/ICST53961.2022.00014

III. D. Tiwari, Y. Gamage, M. Monperrus, and B. Baudry, "PROZE: Generating Parameterized Unit Tests Informed by Runtime Data," *in Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024,
doi: 10.48550/arXiv.2407.00768

IV. D. Tiwari, M. Monperrus, and B. Baudry, "Mimicking Production Behavior with Generated Mocks," *IEEE Transactions on Software Engineering*, 2024,
doi: 10.1109/TSE.2024.3458448

The following contributions by the author, listed in chronological order, are not included in this thesis.

I. L. Zhang, D. Tiwari, B. Morin, B. Baudry, and M. Monperrus, "Automatic Observability for Dockerized Java Applications," *arXiv preprint*, 2021, doi: 10.48550/arXiv.1912.06914

II. D. Tiwari, M. Monperrus, and B. Baudry, "RICK: Generating Mocks from Production Data," *in Proceedings of the 16th IEEE International Conference on Software Testing, Verification, and Validation: Demo track (ICST)*, 2023: pp. 464-466, doi: 10.1109/ICST57152.2023.00051

III. C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, 2023: pp. 5027-5045, doi: 10.1109/TSE.2023.3324950

IV. D. Tiwari, T. Toady, M. Monperrus, and B. Baudry, "With Great Humor Comes Great Developer Engagement," *in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2024: pp. 1-11, doi: 10.1145/3639475.3640099

V. B. Baudry, K. Etemadi, S. Fang, Y. Gamage, Y. Liu, Y. Liu, M. Monperrus, J. Ron, A. Silva, and D. Tiwari, "Generative AI to Generate Test Data Generators," *IEEE Software*, 2024: pp. 1-9, doi: 10.1109/MS.2024.3418570

VI. Y. Liu, D. Tiwari, C. Bogdan, and B. Baudry "An Empirical Study of Bloated Dependencies in CommonJS Packages," *arXiv preprint*, 2024, doi: 10.48550/arXiv.2405.17939

VII. J. Wachter, D. Tiwari, M. Monperrus, and B. Baudry, "Serializing Java Objects in Plain Code," *arXiv preprint*, 2024, doi: 10.48550/arXiv.2405.11294

Acknowledgements

With this most crucial chapter of my thesis, I hope I can convey, even if just the tiniest bit, my deepest appreciation for those who have contributed to it.

I am indebted to the generosity and friendship offered by my stellar supervisors, *Profs.* Benoit Baudry and Martin Monperrus. I believe I struck `faker.preciousMetal()`, having done my PhD under their watchful and humorous [1] guidance. Thanks, Benoit and Martin, for your dedication to your craft, your earnest positivity, and for inviting many like myself to make what we will of the opportunities you give us.

I thank each of my co-authors, from whom I have learned about collaboration and so much more. To all the brilliant and kind members, current and past, of our appropriately named AweSome Software Engineering Research Team (ASSERT), thank you. We gather from distant shores to a little kitchen in our corridor at KTH, with so much in common, and yet such a lot to learn from each other. Belonging to this group has been a source of comfort during the (few weeks of) summers and (rather long) winters of Sweden.

I am eternally grateful to my parents, to whom I owe everything. Thanks also to my family and friends, across time zones, as well as Kaaju, the best good boy on the face of the planet. Thank you for listening politely when I summarized my latest experiments, despite not being asked to.

I would not have had the courage to embark on this adventure, or persevere through it, were it not for Vaibhav. I am immensely proud of the shared interests, the ever increasing number of reruns of our favorite classics, and the two doctoral theses between us, in no specific order. Thank you for our life in Uppsala, which is always worth the commute. I am excited for all that is to come!

I want to express my sincere gratitude to the members of my PhD committee, Professors Paolo Tonella, Mira Mezini, Robert Feldt, and Serge Demeyer. I am thankful to Professor Panagiotis Papadimitrados for serving as advance reviewer of this thesis, and to Professor Aristides Gionis for chairing the proceedings of my PhD defense. My PhD has been generously supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

Contents

Part A

1. Introduction

"Pearl-fishers dive for pearls,
merchants sail in their ships,
while children gather pebbles and
scatter them again. They seek
not for hidden treasures, they
know not how to cast nets."

Rabindranath Tagore,
On the Seashore

Software has established itself as an essential actor in our lives. Our civilization is undeniably reliant on it, and for good reason. The versatility of software has helped cement its place across all enterprises, from finance [2] to culture [3]. We have grown to depend on increasingly more sophisticated software systems to function perfectly — much is at stake if they don't [4]. In order to guarantee their quality and reliability, it is important that software systems are tested meticulously. *Software testing* is the process of verifying that a software system behaves as it is intended to behave. A deviation from an intended behaviour unearths a problem within the system that must be fixed [5].

1.1 Software testing

Everyday, the millions of software developers around the world perform millions of *software tests* [6]. They write *automated* tests to verify that the features added to their software project work as they are required to. Additionally, whenever a bug is discovered and fixed in the system, tests are written to ensure that it is in fact corrected and that it does not reoccur as the project evolves. Depending on the testing practices of its developers, the *test suite* of even modest software projects can contain thousands of automated tests. Each time a new contribution is made by any of its developers, the test suite of a project is run in its entirety. If all tests are green, i.e., all tests verify that the new changes to the project align with expectations and do not introduce unintentional side-effects or *regressions* [7], the project is deployed to production for real-world usage. Production environments are unpredictably diverse, and the same system may behave differently depending on a multitude of factors such as hardware, other software, and end user traffic. Insights from field behaviors, in the form of failure reports [8] and performance

```
1  public class UnitConverter {
2    ...
3    public double convertMilesToKm(double miles) {
4      if (miles <= 0)
5        return 0;
6      return miles * 1.60934;
7    }
8  }
9  ..........................................................................................
10 @Test
11 public void testConvertMilesToKm() {
12   // input
13   UnitConverter converter = new UnitConverter(...);
14   double km = converter.convertMilesToKm(10);
15   // oracle
16   assertEquals(16.0934, km, 0.0);
17 }
```

Listing 1.1: The behavior of the method `convertMilesToKm` is verified by the unit test, `testConvertMilesToKm`. The key components within the test are the test input and the test oracle.

metrics, may be incorporated within the next cycle of development and testing. The process is creative, indispensable, and perpetual.

In this thesis, we focus on the process of creating new automated software tests. An automated software test is a piece of code that runs against other code to establish its validity. Regardless of the scale of the system under test, or the granularity with which its validity is being verified, all tests are composed of two key components. The first component is the *test input*, which brings the system to an initial testable state. This includes setting up the test environment, initializing the program to the desired state, and performing the action to be tested against the system. The second component is the *oracle*, i.e., a concrete specification of the behavior expected from the system under the given test input [9]. The oracle is responsible for automating the comparison between the expected behavior of the system under test with its actual behavior. Developers typically draw from their expertise of the system under test, and their knowledge of the domain, to specify oracles [10]. However, with growing complexity, it becomes increasingly harder to specify the behaviors expected of the system under test [11]. This (in)famous problem in software testing is referred to as the *oracle problem* [12]. We illustrate with the example of Listing 1.1, which presents the automated *unit test*, `testConvertMilesToKm`, for the *focal method* `convertMilesToKm`, which is the method being tested [13]. Within the test, the developer first defines the inputs through program variables, and constructor and method calls. Next, in order to verify the behavior of the focal method, the developer specifies the oracle. The determination of a useful oracle is not a straightforward task. For example, an *implicit oracle* for `convertMilesToKm` is that its invocation should not cause the

program to exit. However, the oracle within `testConvertMilesToKm`, specified through an *assertion* statement, is arguably more meaningful [14], as it is a concrete expression of the expected behavior of `convertMilesToKm`.

The process of writing *good* software tests, which includes designing interesting and representative test inputs and expressing meaningful oracles, requires significant developer effort. It might even be more art than science [15]. Most organisations appreciate this fact, and invest serious resources for the creation and maintenance of strong test suites. However, as highlighted by Wang *et al.* [16,17], in-house developer-written test suites do not account for all the ways in which a system may be exercised by end-users in the field [18]. In fact, envisioning and testing for all scenarios that can occur in the field is hardly possible. As a simple example, the interactions of an end-user with our conversion program from Listing 1.1 might trigger the invocation of `convertMilesToKm` with a value of `-6.6`, causing it to return the output `0`. This scenario within `convertMilesToKm` is not represented by any developer-written test. However, a new test with this combination of test input (`convertMilesToKm` called with `-6.6`) and oracle (output value of `0`) would serve as a useful addition to the existing test suite. Needless to say, for larger, more complex systems, the difference between test and field behaviors is more compelling, motivating the need for automatically addressing this gap.

1.2 Problem statement

When writing automated software tests, developers envision typical usage scenarios for their system based on their understanding of its requirements. These assumptions are encoded into the test suite. However, behaviors expressed within the test suite might not overlap with actual behaviors that occur in the field, as the system executes under real workloads [16,19]. Consequently, real-world usage scenarios that are relevant for end-users are not represented within the test suite. The problem statement we address in this thesis is as follows.

The oracles within developer-written tests often do not reflect behaviors that are exhibited by the system in the field, exercised with real workloads.

1.3 Position within the state of the art

Owing to its necessity and its cost within software engineering, software testing has been the subject of research for decades. A large number of studies contribute techniques for automatically generating tests [11], especially focusing on the generation of test inputs that optimize for *code coverage* [20], such that more of the program is reached by the tests. For example, test inputs can be generated randomly [21], using search-based techniques [22], symbolic execution [23], and with machine learning models [24]. However, there are far fewer studies that aim to assist developers with the oracle problem. The key challenge is to propose

algorithms that can automatically determine the behaviors expected of the system under test. As a solution to this challenge, some works extract oracles from formally or informally documented requirements of the system [25–29]. However, such requirement specifications may not always be available or complete [12].

Addressing the frequent unavailability of structured requirements, another body of work proposes strategies to generate oracles from the observed behaviors of the system. In the absence of specifications, the developer-written test suite can be considered an executable proxy for the requirements, and behaviors can be captured from it to generate new tests [30–34]. The key limitation of this approach is that it, by design, relies on the developer-written test suite which, as with specifications, may not always be available or complete [16]. Moreover, the oracles generated by many of these approaches tend to be simple, and not express complex functionalities of the system under test [35, 36]. Additionally, while valuable, these strategies for automated test generation proposed in the literature do not explicitly aim at bridging the gap emphasized by Wang *et al.* [16] between tested behaviors and field behaviors.

We argue that field executions are a rich source of information that can be harnessed for augmenting the test suite with behaviors that are relevant for real-world users. In this thesis, we derive a solution to the oracle problem [12] by observing systems in the field to determine their expected behaviors.

1.4 Thesis contributions

The contributions made by this thesis are highlighted in red in Figure 1.1. Addressing our problem statement about the differences between test and field behaviors, this thesis contributes with a mechanism for capturing runtime behaviors from the field, and feeding them back into the test suite to augment it. Specifically, we monitor a system as it executes in the field, capturing data corresponding to the target units identified within the system. Next, test generation is triggered in-house, using the captured field data [18]. For the target units, the generated tests contain inputs that reflect the captured production states, and oracles that reflect observed outputs. The generated tests can be appended to the developer-written test suite, and run in-house to verify functional behaviors, and detect regressions within the system as it evolves. We realize three applications of our proposed mechanism, and make the following contributions.

C1 Techniques for deriving oracles from runtime observations for improving test quality
We capture data as systems execute in the field and generate tests with explicit oracles. The oracles contribute to improved test quality, with respect to coverage and mutation analysis.

C2 A technique for deriving mock-based oracles from runtime observations for regression testing
We capture distinct aspects of runtime behaviors, and express them within

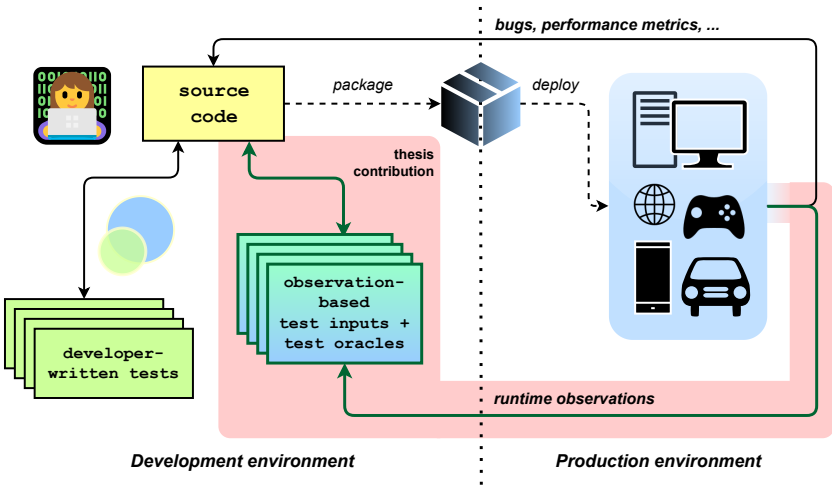


Figure 1.1: The contributions of this thesis (highlighted in red) within software testing. Within the development environment, the developer writes tests, which are run before the project is deployed to the production environment. Feedback from production in the form of bug reports and performance metrics triggers changes to the source code and associated tests. However, some behaviors that occur in production might not be represented within the test suite [16]. This thesis proposes a mechanism for making runtime observations in the production environment, and utilizing them as test inputs and oracles within generated tests. The generated tests reflect real-world usage of the system and can augment the developer-written test suite.

generated tests as mocks, stubs, and mock-based oracles. The generated mock-based oracles mimic field observations and detect regressions.

C3 A technique for generalizing developer-written oracles using runtime observations
We capture the arguments passed to methods at runtime and use them to transform existing unit tests into parameterized unit tests. The developer-written oracles within these parameterized unit tests hold over a larger range of test inputs than envisioned by developers.

1.5 Summary of papers

We now summarize the four research papers included in this compilation thesis, associating them in Table 1.1 with the three contributions listed in Section 1.4.

I. D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, **"Production Monitoring to Improve Test Suites,"** *IEEE Transactions on Reliability*, 2021. doi: 10.1109/TR.2021.3101318

Table 1.1: Contributions made by the papers included in this thesis

Contribution	Paper I	Paper II	Paper III	Paper IV
C1: Oracles for improved test quality				
C2: Mock-based oracles for regression testing				
C2: Generalizing developer-written oracles				

Summary: This paper presents PANKTI, a tool that identifies target methods based on an adequacy criterion defined on the developer-written test suite, captures objects associated with the target methods in the field, and generates unit tests that encode the captured objects as test inputs and test oracles. We evaluate PANKTI against three real-world Java projects, targeting methods that are pseudo-tested by the developer-written test suite. The tests generated by PANKTI augment the developer-written test suite of the three projects we experiment with. A majority of the target methods are no longer pseudo-tested as a result of the explicit oracles derived from runtime observations.

II. L. Zetterlund, D. Tiwari, M. Monperrus, and B. Baudry **"Harvesting Production GraphQL Requests to Detect Schema Faults,"** *in Proceedings of the 15th IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, 2022. doi: 10.1109/ICST53961.2022.00014

Summary: In Paper II, we present AutoGraphQL, a tool that generates tests using incoming HTTP requests for web applications that use GraphQL APIs, defined using a GraphQL schema. The captured GraphQL query requests are replayed within the generated tests. The assertions within the test verify the properties of the response object against the expected properties specified in the GraphQL schema. The tests generated by AutoGraphQL augment the test suites of one open-source and one industrial application, increasing their coverage and detecting faults in the implementation of the schema.

III. D. Tiwari, Y. Gamage, M. Monperrus, and B. Baudry, **"PROZE: Generating Parameterized Unit Tests Informed by Runtime Data,"** *in Proceedings of the 24th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024. doi: 10.48550/arXiv.2407.00768

Summary: We present PROZE in Paper III, as a technique for automatically transforming existing unit tests into parameterized unit tests. PROZE identifies methods that are directly invoked within developer-written unit tests, and observes their invocations at runtime. It uses the union of arguments with which a method gets invoked within the test suite and in the field, to generate an argument provider which supplies values to a unit test parameterized over the target method. We experiment with five Java modules from two large projects, and find that PROZE augments their test suites with parameterized unit tests containing developer-written oracles that in fact generalize over a larger range of test inputs captured at runtime.

IV. D. Tiwari, M. Monperrus, and B. Baudry, **"Mimicking Production Behavior with Generated Mocks,"** *IEEE Transactions on Software Engineering*, 2024. doi: 10.1109/TSE.2024.3458448

Summary: Paper IV introduces RICK, a tool that captures objects corresponding to the invocation of a target method in the field, as well as the method calls that occur within the invocation of the target method. Using the captured data, RICK generates unit tests that mock external objects, stubbing their interactions with the target method. The three types of mock-based oracles generated by RICK verify distinct aspects of observed field behaviors, specifically the output expected from the target method, the arguments with which the mock method calls occur within the invocation of the target method, and the sequence and frequency of these mock method calls. Our evaluation of RICK against three Java applications shows that it augments their test suites with mock-based oracles that complement each other in their ability to detect regressions.

1.6 Thesis outline

Part A presents the contributions of this thesis. Chapter 2 summarizes the literature on oracle generation and observation-based test generation. Next, Chapter 3 introduces our methodological framework, and Chapter 4 describes our experimental results. We share concluding remarks and directions for future work in Chapter 5. Part B contains the four research papers that are included in this compilation thesis.

2. State of the Art

"Boond boond sagar hai,
warna yeh sagar kya hai?"

Javed Akhtar,
Yeh Tara Woh Tara

In this chapter, we discuss the works most closely related to the contributions of this thesis. Section 2.1 summarizes the literature on test oracles, and Section 2.2 presents an overview of automated test generation techniques, focusing on the closely related body of work on test generation based on runtime observations.

2.1 Test oracles

The two key components within each software test, manual or automated, are the test input and the *test oracle*. The test inputs bring the system under test to the desired state, also performing the action to be tested. It is then the job of the oracle to ascertain if the system behaves correctly, i.e., the outcome from the test aligns with expectations.

The ***oracle problem*** [12] refers to the challenge of determining the expected behaviors of the system under test, and verifying that the system conforms to this expected behavior. Finding a solution to the oracle problem is critical for the usefulness of each test [5], yet largely the responsibility of the developer [6, 9]. However, some researchers have contributed techniques for the automated generation, assessment, and improvement of oracles, in an effort to alleviate the oracle problem [37]. In this section, we summarize these studies with respect to two dimensions: first, the determination of the oracle, i.e., the expected behavior of the system that must be checked; and second, the concrete specification of the oracle, such that the conformance of the system with the expected behavior can be automatically verified.

Oracle generation

A large number of studies have explored strategies for the automated generation of oracles, for both developer-written and automatically generated tests. The mechanisms proposed by these studies derive oracles from different sources of

information, such as the documented requirements of a system or its observed behaviors.

Deriving oracles from requirements: Some studies mine requirement specifications, documented in diverse ways, to deduce oracles for the system under test. Li and Offut [25, 38] recommend strategies for defining oracles such that abstract, model-based tests can be transformed into concrete tests that reveal failures. They propose going beyond implicit oracles that check for the absence of runtime exceptions or crashes, and incorporating oracles that observe internal states [39] specified within assertion statements. Carzaniga *et al.* [26] produce oracles for a system using its developer-written Javadoc [40] in conjunction with specifications derived from its intrinsic redundancy, i.e., multiple ways of performing the same action. Given a pair of redundant methods, each generated *cross-checking* oracle, represented as an advice within an aspect class, compares the output of a method call with the result from an equivalent method call. Goffi *et al.* [27] discover that oracles for exceptional behaviors are rarely present in developer-written and automatically generated tests. They propose Toradocu, which uses natural language processing to parse Javadoc comments in order to generate oracles that verify the exceptions thrown by a method. The oracles are implemented as aspects that instrument existing tests, getting activated each time the method is invoked. Blasi *et al.* [28] extend Toradocu into Jdoctor. In addition to the documentation of exceptional behaviors, Jdoctor also parses pre- and post-conditions specified within the Javadoc of methods and constructors. The oracles derived from this analysis are encoded as Java expressions that can be embedded within automatically generated tests [41]. Motwani and Brun [29] present Swami, a technique that generates oracles from specifications that are documented in natural language, without the need for the source code of the project. Swami uses regular expressions for the identification of boundary conditions and exceptional behaviors from the specifications of non-trivial Java, JavaScript, and C++ projects. CrowdOracles [35] delegates the oracle problem to the human participants of a crowd-sourcing campaign. The participants rely on documentation to determine the validity of the assertions within automatically generated tests [22]. A family of tests called *theory-* [42] or *property-based* tests (PBTs) [43, 44] focus on the formally specified or developer-identified properties of a system. The oracles within these tests express properties that must always be true for the system, acting as executable specifications for it [45]. The oracles within PBTs are evaluated against random input data provided by data generators [43].
Reflection: Utilizing the requirement specifications of the system under test is a meaningful strategy for the derivation of oracles that specify the behaviors expected of it. However, by design, this strategy relies on the presence of requirements documented either formally or at least in a semi-structured manner by developers. In practice, there may be limited access to such structured specifications within real-world projects, and these specifications are also subject to

evolution.

Deriving oracles from observed behaviors: The oracles for a system may also be deduced without the use of documented requirements, leveraging observability [46] for oracle generation. The Orstra tool by Xie [47] captures object states during the execution of an automatically generated test suite, which it then uses to add assertion oracles within the tests. These oracles, aptly called *regression oracles*, can detect regressions during subsequent executions of the test suite, as the system evolves [7]. Some studies focus on the values contained in program variables, which can be monitored during the execution of tests and serve as oracle data. Mutation analysis has been used to select the variables used as oracle data, which can be ranked to maximize the fault detection ability of tests for civil avionics [48] and medical device [33] systems. Addressing the generalizability and scalability of this approach, DODONA [32] employs a different strategy. It monitors the interactions of, and dependencies among, variables during test execution, and ranks them using network centrality analysis to recommend oracle data. The goal of these works on oracle data selection is to support developers in their effort to create *expected value test oracles*. Another way of approaching the oracle problem is through metamorphic testing [49], which has also received considerable attention in the literature [50,51]. The oracles used within metamorphic tests specify *metamorphic relations* (MRs), which signify the change in the output from the system with respect to the change in the input [52]. MR-Scout [34] observes the execution of a test suite to capture MRs that encode the change in the state of an object. The MRs are included as assertions within automatically generated tests [22]. Invariants, which are properties that must hold for a system in a specific state, can also serve as oracles. Daikon [30,53] is a popular tool for discovering likely [54] invariants, inferring them from the traces of a system exercised via symbolic execution. DySy [55] extends this approach, inferring invariants through dynamic symbolic execution. A large number of studies on automated test generation encapsulate the invariants detected using these approaches within assertion statements
Reflection: Oracles can be generated automatically, without relying on documented specifications, by considering the observed program behaviors as the expected behaviors. Most contributions in this space capture behaviors by observing the execution of developer-written tests. However, as with specifications, developer-written test suites may not be complete, or even present, in practice.

Generating oracles using learning-based approaches An increasing number of studies perform oracle synthesis using learning-based techniques. ATLAS [14] uses neural machine translation to generate an assertion statement for a test method, given its focal method. The generated assert statements resemble developer-written assertions in their meaningfulness and complexity. Yu *et al.* [56] extend ATLAS by incorporating information retrieval to fetch an assertion for a focal test that is most similar to the test within the training data, and adapting the retrieved assertion for the focal test. Molina *et al.* [57,58] obtain valid pre- and

post-states for a method under test by executing it within generated tests, and then guide a genetic algorithm to learn the postconditions of the method for use as oracles. The inferred oracles, expressed as assertions, encode expected object states. TOGA [59] generates oracles for a focal method, and ranks them using a transformer-based approach. It does not rely on the implementation of the focal method, but rather its signature and documentation, and the prefix of the test generated automatically for it [60]. The generated oracles are expressed as assertions that check the value returned from the method or an exception thrown from it. The works of Hossain *et al.* [61] and Liu *et al.* [62] extend the evaluation [63] of TOGA, highlighting areas of improvement for neural oracle generation. Without the need for an assertion oracle, SEER [64] predicts whether a unit test passes or fails, using neural representations of correct versions of methods embedded with their passing tests, and incorrect versions with their failing tests. The use of Large language models (LLMs) for test oracle synthesis is also being investigated actively [65]. TOGLL by Hossain and Dwyer [66] is a technique for producing diverse assertion and exception oracles by prompting LLMs. The generated oracles have improved fault-detection capabilities, compared to the oracles from TOGA. *Reflection:* As LLMs become more skilled at handling software development tasks, their use for oracle generation is also gaining attention in the literature. This is welcome news for developers who prefer to use LLMs as assistants when writing tests. However, the responsibility of ensuring the validity of their outputs lies largely with developers.

Oracle augmentation

Oracles created by human developers, or those generated automatically, are not guaranteed to be perfect [64]. In their survey, Danglot *et al.* [67] refer to *test amplification* as the process of enriching existing tests, of which oracle improvement is a cornerstone. Several studies assess the effectiveness of oracles, and propose strategies for their augmentation.

Existing oracles differ in their ability to detect faults. Staats *et al.* [68] argue that tests can be prioritized based on their oracles, for quicker fault detection, and thus more effective testing. Their approach for prioritization is based on analyzing the proportion of the program that is actually exercised by the oracle. Schuler and Zeller [69, 70] introduce checked coverage as an indicator for oracle quality. Unlike the traditional coverage metric, which represents the statements that are reached during the execution of the test suite, checked coverage represents covered statements that actually contribute to the results evaluated by the oracles contained in the test suite. Mutation analysis can also help discover inadequate oracles [71], and provide more evidence that coverage does not guarantee fault detection. An extreme example is that of pseudo-tested methods [72]. Despite these methods being covered by existing tests, no oracle detects the replacement of the entire body of these methods with a default return statement. Vera-Pérez *et al.* [73] demonstrate the prevalence of pseudo-tested methods within projects that have test suites with high coverage. OraclePolish by Huo and Clause [74] detects

oracles that over-specify by checking too much, or under-specify by checking too little. An erroneous oracle can raise false alarms (false positives) or fail to detect bugs (false negatives), as discussed by Guo *et al.* [75], who recommend debugging the oracle itself. The OASIs tool proposed Jahangirova *et al.* [76, 77] identifies such deficient oracles, using a search-based approach for the false positive cases, and mutation analysis for the false negative ones. Through an evaluation with a human study [78], the authors also discover that OASIs identifies false positives and negatives more accurately, relative to manual assessment of such cases. Furthermore, GAssert [79, 80] uses an evolutionary algorithm to automatically improve the oracle deficiencies identified by OASIs.
Reflection: The literature on oracle augmentation highlights the inherent complexity of the oracle problem. Oracles are the essence of testing, yet it is challenging to determine meaningful oracles manually, and more so to generate them automatically.

Domain-specific oracles

The oracle problem has been studied in more specific contexts, highlighting its pervasiveness across domains. For example, AGORA [81] adapts the Daikon system [30] to learn oracles that express the invariants of RESTful web APIs, based on their OpenAPI specification and past requests. For the social networking platforms of Meta, the ALPACAS tool [82] generates integrity tests that keep users safe from harmful online content and behavior. Using logs simulated from production-like scenarios, ALPACAS infers oracles which are responsible for decisions regarding user-reported content. Across systems with rich Graphical User Interfaces (GUIs), Augusto [83] can generate oracles that express the outcomes of commonly-used operations, such as those that represent the creation and deletion of entities, or filling up of registration forms. On the other hand, for graphical systems such as digital TVs, the ADVISOR framework [84] facilitates the configuration of computer vision models in order to produce oracles that can visually compare expected and actual outputs.

Many studies focus on automated oracles for mobile applications. For instance, Lin *et al.* [85] argue that taking screenshots of the application under test for use within oracles can overload an already busy Android device. They propose replaying recorded GUI events and photographing the application GUI with an external camera, so that it can be tested using oracles that are automated and device-agnostic. GUI-based oracles that are reusable across applications can also be derived from common interactions with UI elements and their corresponding response [86, 87]. Jabbarvand and colleagues [88] approach the oracle problem for Android applications from the perspective of their energy consumption, employing a deep learning model to detect unusual patterns of energy usage.

Several studies address the oracle problem for cyber-physical systems, big and small [33, 48]. For example, Gay *et al.* [89] propose steering the models used to test real-time embedded systems to allow for more flexible oracles. These oracles can accommodate the non-deterministic behaviors that are characteristic of such

systems, while successfully verifying acceptable behaviors. Mithra by Afzal *et al.* [90] uses simulated telemetry data for cyber-physical systems to learn oracles for them that detect anomalous behavior. Oracles for elevators can be generated and specified in domain-specific languages so that they can be continually evolved and tested [91], and machine learning models can also provide oracles that help identify bugs in their functionalities [92]. Jahangirova *et al.* [93] propose oracles for self-driving cars, deriving them from metrics that represent the quality of driving of human drivers.

Reflection: Finding a solution to the oracle problem is a challenge across all domains. Much research has been done on automatically determining the expected behaviors for diverse kinds of systems. The strategy used is typically fine-tuned for the system under test, and there is no silver bullet for automatic oracle determination.

2.2 Automated test generation

There are several studies that seek to aid developers in the arduous and essential task of writing software tests [11]. Section 2.1 has summarized the literature on the automated generation and augmentation of oracles. We now present studies that propose mechanisms for the automated generation of whole tests, including their inputs and their oracles.

Test inputs can be generated randomly, such as with well-known Randoop [21, 41] that produces a sequence of method calls, each call appended after feedback from the preceding call sequence. Using the execution of this sequence, in conjunction with specifications for the classes under test, Randoop produces oracles that express expected and erroneous behaviors. Several tools combine symbolic and concrete executions, into a technique called dynamic symbolic execution or *concolic* testing [11]. Examples include DART for C programs [94], CUTE for C [23] and its counterpart jCUTE for Java programs [95], and PEX for .NET [96]. These tools use constraint solvers to generate inputs by assigning concrete values to symbolic variables, while maximizing code coverage. The oracles used within the generated tests are implicit, they check for crashes or violations of existing assertions. Fraser and Zeller propose μTEST [71], which relies on mutation analysis of an existing test suite, to generate new unit tests containing oracles that kill surviving mutants. This approach has since been incorporated into EvoSuite [22, 60], a popular automated test generation technique. EvoSuite uses an evolutionary search algorithm to generate test inputs, optimizing for code coverage. The generated oracles are further minimized, while killing the maximum number of mutants. Not just oracles (cf. Section 2.1) but complete tests can also be derived from specifications. Liu and Nakajima [97] generate tests that verify the conformance of the implemented version of the system with its formal specifications.

Reflection: These techniques are useful for automatically generating inputs that result in high test coverage. However, several studies [15, 20, 36] advocate for the representativeness [19, 97] of complex objects used as test inputs [98, 99],

explicit oracles that are sensitive to faults, as well as better overall understandability of automatically generated tests [100, 101]. Moreover, their generation effort targets high structural coverage, and does not address the behavioral differences between test and field executions [16].

Observation-based test generation: Most closely related to the contributions of this thesis are the relatively fewer works on test generation based on runtime observations. To motivate the need for observation-based testing, many of these studies cite the lack of overlap between behaviors that are tested in-house and behaviors that are exhibited by the system in the field. This is also the observation of Wang *et al.* [16], who demonstrate that field behaviors cover more code and kill more mutants than in-house tests. Wang and Orso [17] follow up on these findings in a short paper that proposes a technique called Replica for mimicking user behaviors within tests. For untested behaviors, Replica captures field data and supplements it with similar inputs produced through symbolic execution. Invariants for the system serve as the oracles, and the evaluation is based on the invariants covered by in-house tests versus those covered by the field-based inputs. *Capture (or record) and replay* is a related area of research. The premise of this approach is to monitor the sequence of events that occur as a system executes, and replay it offline so that the system is brought to the same state within the test. jRapture [102] is a capture and replay tool that logs the interactions between the target application and its host system, and replays them within tests. GenUTest [103] observes the sequence of method calls to generate observation-based tests that use mocks. However, most capture and replay techniques typically emphasize on the captured test inputs, but not the oracle [85]. The related technique of *test carving* refers to the extraction of unit tests from larger tests. Elbaum and colleagues [104] carve unit tests from slow-running system tests. Kampmann and Zeller [105] carve out parameterized unit tests from system tests. MicroTestCarver proposed by Deljouyi and Zaidman [101] produces unit tests from manual or automated end-to-end tests. A key limitation of the test carving approach is its reliance on system tests. Behaviors that are not represented within the system tests will also not be represented within the generated tests.

Bertolino *et al.* [18] survey 80 studies that propose strategies for field-based testing, grouping them into three categories. First, *ex-vivo* techniques use data from the field, for use within in-house tests inside the development environment [106]. Second, with *offline in-vivo* techniques, tests are run on a clone or a fork of the system, within the production environment [107]. Third, *online in-vivo* techniques can be likened to chaos engineering [108], and involve running tests directly on the target system in production, in parallel with normal execution [109–111]. A key finding from this survey is that the oracle problem is often overlooked among all three categories. They identify this as an essential direction for research on the topic.

Recently, automated test generation from field observations has been adopted

by Meta [112]. Closely aligned with our work, their approach, called TestGen, serializes the objects related to the invocation of methods, and uses them to generate regression tests. For non-void methods, the oracles within the generated tests verify the values observed in the field. For methods that do not return values, the oracles verify the absence of exceptions. Over a period of six months, the tests generated by TestGen have detected regressions in the codebase of Instagram.

Reflection: The fact that field behaviors differ from tested behaviors is well understood within testing literature. Several studies propose field-based testing approaches, yet most focus on capturing inputs from the field. Automated oracle generation from production observations is under-represented among these studies. The generation of complete tests from field data – including inputs and oracles – has practical advantages [19].

3. Methodological Framework

"That rug really tied the room together."

The Big Lebowski

In this chapter, we present our approach for observing runtime behaviors with the goal of harnessing oracles that reflect the observed behaviors. First, Section 3.1 describes our key technical contribution, a framework that leverages runtime observations within test oracles. Next, in Section 3.2, we introduce the research questions we address in this thesis. Section 3.3 summarizes the protocol we use to answer our research questions, and Section 3.4 gives an overview of the projects we conduct our experiments on.

3.1 Framework for augmenting test oracles with runtime observations

In order to bridge the gap between tested behaviors and field behaviors, we propose a novel conceptual framework that observes runtime behaviors with the goal of utilizing them within tests. As illustrated in Figure 3.1, our framework receives three inputs related to a project: its source code, its test suite, and a runtime *workload* with which it can be exercised. The term workload refers to a set of

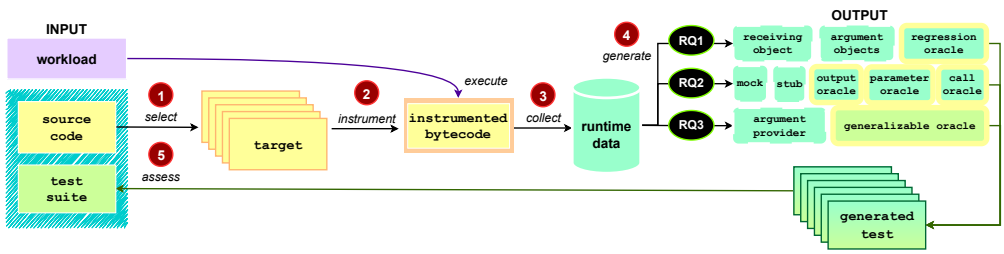


Figure 3.1: The general framework proposed in this thesis for augmenting test oracles with runtime observations. When the input project is exercised with a workload, our framework captures data at runtime and utilizes it to generate tests. We initialize this framework in distinct ways to answer our research questions (introduced in Section 3.2).

representative operations that can be performed against a running system. If the project contains a test suite, the workload can refer to the execution of the tests in the test suite [113,114]. In the context of this thesis, we utilize workloads that exercise the system in the field [16,18], such as fetching directions from a routing application, or loading, editing, and saving a text file with a document processing program. We rely on field workloads motivated by the need to generate tests that are representative of field behaviors, as highlighted by Wang *et al.* [16]. The output from the framework is a set of focused tests for the project that contain an explicit oracle informed by runtime behaviors.

Test target selection: Our proposed framework works as follows. Phase (1) involves using the input source code to define a subset of target components that the generated tests will be associated with. These components may be selected based on developer-defined criteria. For example, the set of methods that have recently been updated make ideal candidate targets [115]. Alternatively, we may deem as targets the methods in the source code that are inadequately tested, based on coverage [116,117] or mutation analysis [118,119] of the input test suite. It is essential to define a clear subset of methods to target for a focused monitoring campaign, that adds minimal overhead to the execution of the system at runtime.

Code instrumentation: Once the target methods are identified, phase (2) focuses on instrumenting them. The goal of the instrumentation is to add additional instructions that allow us to monitor the invocations of each target method, and configure what will be persisted at runtime, and how. The information that is available for us to harness at runtime is plentiful. For instance, we can monitor and store the sequence of method invocations [120] within a developer-defined class, and save snapshots of the objects associated with a target method [112]. We realize these requirements through dynamic instrumentation, without modifying the source code of the target methods, by preparing an agent. The agent includes instructions that are triggered when an event occurs at runtime. The agent is attached to the project when it is being deployed, such as through a `javaagent` for Java projects.

Runtime data collection: In the subsequent phase (3), the instrumented version of the program is executed with the input test and/or field workload. As we see from Figure 3.1, running the workload against the instrumented version of the program leads to the collection of runtime data on disk. Each invocation of a target method triggers the instructions within the instrumentation agent, which is configured with the exact protocol to be employed for data collection. For instance, the runtime data may be persisted on to a database, or within files, serialized with the help of serialization libraries [121] in binary format, or as arguably more readable XML or JSON [122]. Regardless of the protocol employed for the serialization, it is useful to persist related objects such that they can be grouped together. For example, assigning identifiers to the receiving, argument, and returned objects can allow us to associate them with a single invocation of a target method. In this phase of data collection also, configurations made within

the instrumentation agent can help minimize the overhead introduced from the serialization. One way to achieve this could be to set an upper limit on the number of instances of an event, such as a method invocation, to persist, or to save runtime data worth a certain amount of disk space [112].

Test generation: Eventually, after capturing the required kind and quantity of runtime data, the developer induces phase (4) offline (in-house) [18]. Based on the predetermined goal of the observation and data collection campaign, it is in this phase that the runtime data is utilized. Again, we can program the runtime data to be utilized in different ways. Figure 3.1 presents the three sets of artifacts, expressed as code elements, that we utilize for each of the research questions outlined in Section 3.2. The first set illustrates that the serialized receiving and argument objects can be utilized within a generated unit test to set up the test inputs, while the returned object can be encapsulated within an assertion statement, serving as a regression oracle. We explore the tests resulting from these artifacts in RQ1. Next, runtime data corresponding to the invocation of a target method, as well as nested method calls within this invocation, can be used to mock objects and stub them. Distinct aspects of their interactions can then be verified through dedicated *mock-based* oracles, which we discuss in RQ2. Alternatively, the set of arguments with which a target method is invoked can also be encapsulated within an argument provider method, to be used by a parameterized unit test [123]. RQ3 describes how developer-written oracles can be utilized within generated parameterized unit tests. In all three cases, the code artifacts produced are arranged into unit tests, which constitute the outputs from this crucial phase of our framework. These tests are compilable and executable, and can readily be appended to the existing test suite.

Test quality assessment: Finally, phase (5) involves incorporating the tests output from the previous phase into the test suite of the target project. Here, depending on the initial criteria used to select the target methods, an assessment can be performed to confirm that the new tests are useful additions. The generated tests contain inputs and oracles derived from the behaviors exhibited by the system at runtime, running against the input workload. Therefore, by design, they encapsulate representative behaviors that can complement developer-written unit tests.

Using the framework in the software development lifecycle: The framework of Figure 3.1 can be instantiated in different ways, depending on the specific goal of the observation-based test generation effort. For example, the component responsible for selecting targets can be tuned to cater to developer-defined criteria, and the instrumentation agent can be configured to capture the runtime data as required. In order to minimize the performance costs incurred from capturing data at runtime, the framework can be employed periodically, when modifications are made to the project. In this thesis, we demonstrate the application of the framework for two use cases. First, developers can utilize the framework during manual, alpha-testing campaigns to ensure that mission-critical operations are

represented in the generated tests. Second, for beta-testing campaigns in the field, the framework can be deployed to capture runtime behaviors triggered by real users, after ensuring that their privacy is not violated and sensitive data is not captured. In either case, the generated tests can be integrated into the test suite to complement the existing developer-written tests, or to bootstrap the creation of a test suite if one does not exist, as we do in Paper II.

3.2 Research questions

This thesis examines the implications of incorporating oracles that express observed runtime behaviors into the test suite of a software project. Our work addresses the following research questions.

RQ1 To what extent do oracles derived from runtime observations contribute to improved test quality?
This RQ assesses the extent to which observing a system as it executes, and harnessing these observations within automatically generated oracles, can complement the developer-written test suite of the system, per a specific adequacy criterion.

RQ2 To what extent do mock-based oracles derived from runtime observations contribute to regression testing?
The goal of this RQ is to determine whether different aspects of runtime behaviors can be faithfully recreated and verified within generated tests that make use of mocks. We also analyze the effectiveness of mock-based oracles in detecting regressions within the system.

RQ3 To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?
With this RQ, we analyze whether the oracles within developer-written unit tests are valid over a larger set of inputs captured while observing the system at runtime.

3.3 Experimental protocols

We initialize the general framework introduced in Section 3.1 to answer our three research questions listed in Section 3.2. This section describes our experimental protocol for each research question, applying the framework against the study subjects described in Section 3.4. For each question, we describe the criteria used to select the test generation targets, the runtime data captured for these targets, the output from the framework, as well as the methodology employed for the assessment of the generated tests. These four aspects are also summarized in Table 3.1.

Protocol for RQ1

We answer RQ1 by initializing our general framework in a tool called PANKTI, evaluating it against three study subjects exercised with their field workloads as

detailed in Section 3.4, PDFBox, BroadLeaf, and Jitsi. Note that this protocol can be adapted to target other source artefacts for the automated generation of tests. For example, we design a tool called AutoGraphQL that generates unit tests by capturing incoming requests that arrive at the GraphQL API endpoint, and oracles using the developer-defined GraphQL schema of web applications. Paper II describes AutoGraphQL and its evaluation in detail. In this thesis, we answer RQ1 using the evaluation conducted with pankti in Paper I.

Targets: The target methods for test generation are based on an adequacy criterion based on mutation analysis, defined on the original test suite of each project. Specifically, as indicated in Table 3.1, we target methods that are *pseudo-tested* [72, 73] for our evaluation. Such methods are covered by the test suite, meaning that they are invoked at least once during the execution of the existing tests. However, when the body of a pseudo-tested method is replaced by a statement returning a default value, none of the tests that cover the method detect this extreme mutation and fail. This indicates the lack of an explicit oracle for verifying the output of this method, rendering them inadequately tested. For example, the `PlanetExpress` class of Listing 3.1 defines two methods `calcBaseFare` and `makeReservation`. A developer-written test `testThatReservationIsSuccessful` indirectly covers `calcBaseFare`, through the invocation of `makeReservation` on line 19. However, removing the original body of `calcBaseFare` (lines 3-4) and simply returning the value `0.0` (line 5) from it does not cause the test to fail. This makes `calcBaseFare` pseudo-tested, and therefore a target for test generation with pankti.

Runtime data: Per the framework of Figure 3.1, pankti instruments each target method, adding instructions that facilitate the persistence of associated runtime data. Exercising the system with a workload in the field triggers the invocation of the target pseudo-tested methods. Corresponding to each of these invocations, pankti captures an *object profile*, which includes the object on which a target method is invoked, i.e., its *receiving object*, the objects that are passed to this invocation as the arguments, as well as the object returned from this invocation. pankti serializes these objects as XML, and saves them to disk.

Output: The captured object profiles are utilized to generate unit tests for each target method, that reflect the behaviors observed at runtime. For our example of `calcBaseFare` in, the test generated by pankti is presented on lines 22 to 33 of Listing 3.1, using the receiving, argument, and returned objects captured for one invocation of the pseudo-tested method. Each test generated by pankti follows the *Arrange-Act-Assert* pattern [124]. The arrange phase recreates the receiving object (line 26) and the arguments, deserializing them from their serialized representation. In the act phase of the test (line 29), the target method is invoked on the recreated receiving object, passing it the recreated arguments. Finally, the oracle in the generated test verifies, through the assertion statement on line 32, that the output from this invocation is equal to the object returned at runtime. The unit test generated for each target method also serves as a regression test, expressing its behavior as observed at runtime, and capable of detecting

```
1  public class PlanetExpress {
2    public double calcBaseFare(String source, String destination) {
3      ...
4      return fare;
5      return 0.0;
6    }

8    public ReservationStatus makeReservation(...) {
9      ...
10     double fare = calcBaseFare(source, destination);
11     return ...;
12   }
13 }
14 ..........................................................................................
15 // Developer-written test that covers calcBaseFare(String, String)
16 @Test
17 public void testThatReservationIsSuccessful() {
18   ...
19   assertEquals(..., planetExpress.makeReservation(...));
20 }
21 ..........................................................................................
22 // Test generated automatically by PANKTI  for calcBaseFare(String, String)
23 @Test
24 public void testForCalcBaseFare() {
25   // Arrange
26   PlanetExpress planetExpress = deserialize(
         "<pe.planet.express>...</pe.planet.express>");

28   // Act
29   double fare = planetExpress.calcBaseFare("Earth", "Moon");

31   // Assert
32   assertEquals(42.42, fare, 0.0);
33 }
```

Listing 3.1: **(RQ1)** The method `calcBaseFare` defined in the class `PlanetExpress` is covered by the test `testThatReservationIsSuccessful`, but is pseudo-tested. PANKTI generates the test `testForCalcBaseFare` for `calcBaseFare` using runtime observations, as a result of which it becomes well-tested.

changes to this behavior as the project evolves.

Assessment: We analyze the usefulness of the tests generated by PANKTI with respect to the original adequacy criterion. Each target method which is no longer pseudo-tested as a consequence of the generated tests represents a successful case. In our example, the extreme mutation of `calcBaseFare` will cause `testForCalcBaseFare` to fail, making the method well-tested. We present the results for this first RQ in Section 4.1.

Protocol for RQ2

A *mock* is fixture used within a unit test to replace an actual object with a skeletal implementation [125]. Mocking allows for more focused testing, removing side effects from operations on external objects [126, 127]. The behaviour of mocks is configured using *stubs* [128]. Stubbing is a mechanism for returning outputs from a mock when it is triggered with a specific input, bypassing an actual operation. In order to answer RQ2, we initialize our framework of Figure 3.1 into a technique called RICK, which generates unit tests with mocks, stubs, and *mock-based* oracles. Paper IV presents the evaluation of RICK against GRAPHHOPPER, GEPHI, and PDFBOX, exercised with the field workloads described in Section 3.4.

Targets: Note from Table 3.1 that the targets for test generation with RICK are methods that invoke *mockable* methods. A mockable method refers to a method that is called within the body of a target method, on a parameter or field object of a type that is external to the declaring type of the target method. We illustrate this through the example of the target method `createIssue` defined in the class `Catalog` in Listing 3.2. This target method calls the method `register` on an object of type `AdService` (line 5), which is declared as a field within `Catalog` (line 2). Additionally, `createIssue` calls two methods, `getAvailability` (line 7) and `dispatch` (line 8), on the object of type `PrintService`, which is one of the parameters of `createIssue` (line 4). This makes these three nested method calls to `register`, `getAvailability`, and `dispatch` mockable within the target method `createIssue`.

Runtime data: Per Table 3.1, RICK captures data about each target method, and its corresponding mockable methods, by instrumenting them to monitor their invocations at runtime. As with PANKTI, for one invocation of a target method, RICK captures its receiving object, its arguments, as well as the object it returns. Additionally, for each mockable method invocation that occurs within the invocation of the target method, RICK serializes the arguments, the returned object, as well as the sequence and frequency with which these nested calls occur.

Output: Using the data captured at runtime, RICK generates unit tests that mock the external types, stub their behavior, and contain three kinds of mock-based oracles. First, the *output oracle*, OO, verifies the equality of the output returned from the target method within the test with the returned object captured in the field. Next, the *parameter oracle* or PO verifies the arguments with which the mock method calls occur within the target method. The third oracle is the *call oracle* (CO), which verifies the order and frequency of these mock method calls

```
1  public class Catalog {
2    AdService adService;
3    ...
4    public boolean createIssue(String theme, PrintService service) {
5      adService.register(theme); // mockable method #1
6      ...
7      if (service.getAvailability() >= 1) { // mockable method #2
8        service.dispatch(numIssues); // mockable method #3
9      }
10     ...
11     return issueStatus;
12   }
13 }
```

Listing 3.2: **(RQ2)** The target method `createIssue` contains three mockable method calls on objects of external types.

within the target method. Each mock-based oracle thus verifies a unique aspect of the captured runtime behavior. Listing 3.3 presents the tests generated by RICK for the target method `createIssue` of Listing 3.2, using the observations made from one of its invocations in the field. Note that the arrange and act phases (lines 3 to 10) of Listing 3.3 are common to the three generated tests, `testForCreateIssue_OO` (lines 16-21), `testForCreateIssue_PO` (lines 23-30), and `testForCreateIssue_CO` (lines 32-40). Each generated test first reconstructs the receiving object by deserializing it (line 4). This is followed by statements that mock the external field (line 5) and parameter (line 6) types. Next, the mocks are configured by stubbing them using observed inputs and outputs. The statement on line 7 stubs `mockPrintService` to directly return `3` when `getAvailability` is invoked on it, without the actual invocation of `getAvailability`. The target method is then invoked on the receiving object in the act phase of the generated tests, as on line 10. Finally, each mock-based oracle, OO, PO, and CO, verify the target method output, the arguments of the mock method calls, and the sequence and frequency of these calls, respectively.

Assessment: In order to answer RQ2, and determine if production observations can be faithfully recreated through mocks, we execute the tests generated by RICK. Each passing test represents a successful case, where RICK has correctly captured field behaviors, and expressed them within the generated test, including the reconstruction of the objects, the configuration of the mocks through stubs, as well as the assertion and verification statements that constitute the mock-based oracles. Furthermore, we conduct mutation analysis [129] to assess the effectiveness of the three kinds of mock-based oracles produced by RICK at detecting regressions in the system. Section 4.2 presents the results from this evaluation.

Protocol for RQ3

To answer RQ3, we initialize the framework of Figure 3.1 into a mechanism called PROZE, evaluate it using the test and field workloads of PDFBox and SAML

```
@Test
public void testForCreateIssue_XO() {
  // Arrange
  Catalog catalog = deserialize("<jpeter.man.catalog>...</jpeter.man.catalog>");
  AdService mockAdService = mockField_adService_InCatalog(catalog);
  PrintService mockPrintService = mock(PrintService.class);
  when(mockPrintService.getAvailability()).thenReturn(3);

  // Act
  boolean status = catalog.createIssue("HATS", mockPrintService);

  // Assert
  ...
}
.....................................................................................
@Test
public void testForCreateIssue_OO() {
  ...
  // Assert
  assertTrue(status);
}
.....................................................................................
@Test
public void testForCreateIssue_PO() {
  ...
  // Assert
  verify(mockAdService, atLeastOnce()).register("HATS");
  verify(mockPrintService, atLeastOnce()).getAvailability();
  verify(mockPrintService, atLeastOnce()).dispatch(42);
}
.....................................................................................
@Test
public void testForCreateIssue_CO() {
  ...
  // Assert
  InOrder orderVerifier = inOrder(mockAdService, mockPrintService);
  orderVerifier.verify(mockAdService, times(1)).register(anyString());
  orderVerifier.verify(mockPrintService, times(1)).getAvailability();
  orderVerifier.verify(mockPrintService, times(1)).dispatch(anyInt());
}
```

Listing 3.3: **(RQ2)** Using runtime data captured from the invocations of the target method `createIssue` and its mockable method calls (Listing 3.2), RICK generates unit tests with mocks, stubs, and mock-based oracles: OO, PO, or CO.

(introduced Section 3.4), and present the evaluation in Paper III. PROZE transforms existing developer-written unit tests into *parameterized unit tests* [123]. A parameterized unit test (PUT) is a kind of unit test that accepts at least one parameter. The oracle within a PUT is comparatively more general, as it holds for all values supplied to its parameters [130]. When a PUT is run, its parameters are sequentially assigned values by an associated *argument provider*, which is typically implemented as a method.

Targets: As noted in Table 3.1, PROZE selects as targets the methods that are called directly by a developer-written unit test. Consider the developer-written unit test called `testOneGalaxy` presented in Listing 3.4. The test does not have any parameters, and contains three assertion statements. The input within this test includes the invocation of the method `locateObject` with the `String` argument `M51a` (line 4). PROZE identifies the `locateObject` method as a target as it is directly invoked within the test.

Runtime data: PROZE instruments each target method, with the goal of capturing the arguments that are passed to it whenever it is invoked during the execution of the test suite, or in the field as a consequence of the field workload. For example, during the execution of all the developer-written tests, PROZE finds that the target method `locateObject` is invoked with two unique `String` arguments, `M51a` (the original argument passed to it within `testOneGalaxy`), and `M63`. Additionally, PROZE observes that `locateObject` is invoked with the arguments `M100`, `M101`, and `M106` in the field. PROZE captures all five of these arguments.

Output: PROZE generates the argument provider method, using the set of arguments captured for a target method at runtime, across test and field executions. Each generated PUT is parameterized over a single target method. The arguments passed to the PUT by the argument provider are assigned as the arguments to the target method. Moreover, as is true for `testOneGalaxy` in Listing 3.4, a developer-written unit test may have multiple oracles, expressed through multiple assertion statements. In order to discover which of these oracles is generalizable to a wider input range, PROZE isolates each oracle into its own PUT. Lines 10 to 16 of Listing 3.4 presents one of the PUTs `testMultipleGalaxies` generated by PROZE from `testOneGalaxy`, as well as its argument provider method `provideGalaxyID` (lines 18 to 27). The PUT has a single parameter `identifier` of type `String` (line 12), which is assigned as the argument of the target method `locateObject` (line 14). Note that the PUT contains only one of the three assertion statements of the original test (line 15). The argument provider method includes all the five `String` values captured during test and field invocations of `locateObject`, including the argument in the original test, `M51a`. When `testMultipleGalaxies` is run, it is supplied with each of these five values, and the assertion statement is evaluated against each of them.

Assessment: By design, each test generated by PROZE contains a developer-written oracle. In order to answer RQ3 and determine if runtime observations can facilitate the generalization of existing developer-written oracles, we run these generated tests. As mentioned in Table 3.1, we classify the oracle of a PUT as

```java
@Test
public void testOneGalaxy() {
  MessierCatalog catalog = MessierCatalog.initialize();
  MessierObject object = catalog.locateObject("M51a");
  assertEquals("Whirlpool", object.getName());
  assertEquals(CANES_VENATICI, object.getConstellation());
  assertTrue(catalog.getSpiralGalaxies().contains(object));
}
.....
@ParameterizedTest
@MethodSource("provideGalaxyID")
public void testManyGalaxies(String identifier) {
  MessierCatalog catalog = MessierCatalog.initialize();
  MessierObject object = catalog.locateObject(identifier);
  assertEquals(CANES_VENATICI, object.getConstellation());
}

// Argument provider for the PUT
private static Stream<Arguments> provideGalaxyID() {
  return Stream.of(
    Arguments.of("M51a"),
    Arguments.of("M63"),
    Arguments.of("M100"),
    Arguments.of("M101"),
    Arguments.of("M106")
  );
}
```

Listing 3.4: **(RQ3)** A developer-written unit test, `testOneGalaxy`, calls the target method `locateObject` and contains three assertions to verify the details of a single test input. PROZE reuses the second assertion within the generated parameterized unit test `testManyGalaxies`, which is parameterized over `locateObject`. The `String` parameter of this test is supplied values captured at runtime, by the argument provider `provideGalaxyID`.

being *strongly-coupled*, *decoupled*, or *falsifiably-coupled*. The oracle of a PUT is strongly-coupled if it holds only for the original test input, and no other input supplied by the argument provider. On the other hand, if the PUT passes for all inputs supplied to it by the argument provider, we conclude that the oracle is decoupled with the target method. An oracle is falsifiably-coupled if it is valid for a subset of the arguments supplied by the argument provider, while failing for the others. This indicates an oracle that is demonstrably able to distinguish between valid and invalid inputs. For example, the oracle, and consequently the PUT, of Listing 3.4 passes for three of the five inputs, since `M51a`, `M63`, and `M106` all belong to the same constellation of `CANES_VENTICI`, whereas the two others do not. This means that we have succeeded in discovering an existing oracle that is generalizable, as well as falsifiably-coupled with the input supplied to the target method `locateObject`. We present the details of our evaluation of PROZE, and the answer to RQ3 in Section 4.3.

Table 3.1: Summary of our experimental protocol for RQ1, RQ2, and RQ3. We initialize the infrastructure of Section 3.1 to select different kinds of Targets, and capture diverse Runtime Data, with the goal of producing the Output that addresses the three RQs, using relevant Assessment metrics.

	Targets	Runtime data	Output	Assessment
RQ 1	pseudo-tested methods	receiving object, argument objects, returned object	unit tests with **regression oracle** verifying returned object	method status: pseudo-tested vs. well-tested
RQ 2	methods and mockable methods called by them	for methods: receiving object, argument objects, returned object, and for mockable methods: argument objects, returned object, method call sequence	unit tests with mocks, stubs, and mock-based oracle **(output oracle, parameter oracle, call oracle)**	faithful reproduction of production behavior, and mutation analysis
RQ 3	methods called directly by developer-written unit tests	argument objects	argument providers and parameterized unit tests with **generalizable, developer-written oracle**	strongly-coupled, decoupled, and falsifiably-coupled oracles

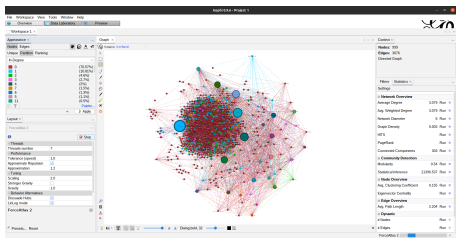
Summary of experimental protocol

Table 3.1 highlights the four key aspects of the protocol we use to answer our three research questions.

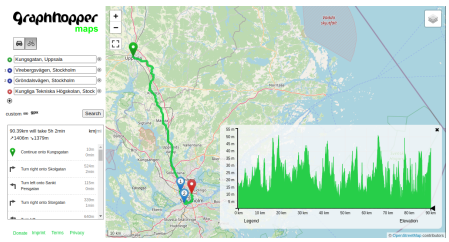
3.4 Study subjects and workloads

In order to answer our three research questions, we conduct experiments on real-world software projects, exercising them with representative field workloads. Designing these workloads is an essential aspect of our evaluation, since the runtime observations we utilize for test generation are sourced from them. In this section, we list these projects, in alphabetical order, presenting their summary and describing their workloads.

1. BroadLeaf: An open-source Java project, BroadLeaf is an e-commerce application implemented using the Spring Boot framework. It includes fea-



(a) Gephi



(b) GraphHopper

(c) Jitsi

Figure 3.2: Snapshots from the field — we exercise the study subjects with realistic workloads in order to capture data that reflect their runtime behaviors.

tures for product, user, and order management. As of this writing, BroadLeaf has over 80 contributors on GitHub. For our experiments with BroadLeaf in the field for Paper I, we design a representative e-commerce workload, running its server and interacting with its front-end.

2. Frontapp: We use Frontapp as an industrial study subject for our experiments in Paper II. Frontapp is the primary, user-facing website developed and maintained by RedEye AB, an investment banking company based in Stockholm. Its backend server, developed in PHP, handles requests through both REST and GraphQL API endpoints. The runtime data we captured for our experiments were GraphQL query requests, triggered by actual user traffic over a period of 33 days.

3. Gephi: A GUI application for working with graph data, the Gephi project has nearly 90 contributors on GitHub. We use Gephi for our evaluation in Paper IV. For our experiments, we import a graph dataset of the artifacts published on Maven Central [131] into a running instance of Gephi, compute statistics on it, such as its average degree and density, manipulate its layout (pictured in Figure 3.2a), and export the resulting graph as PDF and SVG.

4. GraphHopper: Well-known and open-source, GraphHopper is a Java application for fetching the route between different coordinates on a map, using diverse modes of transportation. The repository of the project on GitHub has

over 100 contributors. We use GraphHopper for our experiments in Paper IV, requesting for the car and bike routes between several locations in Sweden (as depicted in Figure 3.2b).

5. Jitsi: Jitsi is an open-source video-conferencing platform implemented in Java. The project repository on GitHub has nearly 500 contributors. We use Jitsi, monitoring targets within its `jicofo` component, as a study subject in Paper I, conducting an hour-long video call among the authors, also sending messages in the chat box (Figure 3.2c).

6. PDFBox: A project from the Apache Software Foundation, PDFBox is a library and command-line tool for PDF manipulation. It supports most PDF generation and manipulation operations, including conversion from and to text and images, encryption, or merging and splitting, among others. We use PDFBox as a study subject in Papers I, III, and IV, using it to perform 10 representative operations on 5 real-world documents from [132].

7. SAML: An extension of the WSO2 identity server, SAML provides functionalities for performing user authentication using the Security Assertion Markup Language (SAML) protocol. The project has nearly 100 contributors on GitHub. For our evaluation in (Paper III), we work with two modules of SAML, called `query` and `sso`, signing in and out across different applications.

8. Saleor: Saleor is an e-commerce application with a backend server implemented in Django, a frontend based on TypeScript, and a GraphQL API to handle HTTP requests. More than 250 developers have contributed to the GitHub repository of Saleor. For our experiments with it in Paper II, we interact with its frontend, performing actions that are typical of e-commerce platforms for both users and administrators.

4. Experimental Results

"There is nothing like looking,
if you want to find something."

J.R.R. Tolkien, The Hobbit

This chapter presents the answers to our three research questions, per their protocol detailed in Chapter 3. We conclude this chapter with a summary of our findings in Section 4.4.

4.1 RQ1: To what extent do oracles derived from runtime observations contribute to improved test quality?

Per the protocol described in Section 3.3, we initialize our general framework explained in Section 3.1 into a test generation pipeline called PANKTI. We evaluate PANKTI against BROADLEAF, JITSI, and PDFBOX, exercising them with the representative field workloads outlined in Section 3.4. We rely on extreme mutation analysis with Descartes [73] for selecting the targets for test generation with PANKTI. Descartes reports the methods that are pseudo-tested by the developer-written test suite of each project.

Table 4.1: Summary of results for RQ1: PANKTI generates TESTS for pseudo-tested TARGETS using observations made at runtime, as the three PROJECTS run in the field. Of these 86 target methods, 61.6% are no longer pseudo-tested, as a consequence of the oracles generated by PANKTI.

PROJECT	TARGETS	TESTS	PASSING	FAILING	TARGET STATUS	
					PSEUDO-TESTED	WELL-TESTED
BROADLEAF	11	351	244	107	5 / 11	**6 / 11**
JITSI	29	20	20	0	10 / 29	**19 / 29**
PDFBOX	46	13,851	13,614	237	18 / 46	**28 / 46**
TOTAL	86	14,222	13,878 (97.6%)	344 (2.4%)	33 / 86 (38.4%)	**53 / 86 (61.6%)**

Table 4.1 summarizes the results from our evaluation. In total, we target 86 pseudo-tested methods across the three projects, which include 11 methods in BroadLeaf, 29 in Jitsi, and 46 in PDFBox. Recall from the example of Listing 3.1 that these methods are covered by at least one test, but their behavior is insufficiently specified by all existing developer-written oracles. PANKTI instruments each of these 86 target methods in order to monitor their invocations at runtime, triggered by the field workload of the system. Corresponding to each invocation, PANKTI captures the receiving object, the argument objects, as well as the object returned from the invocation. These objects are serialized to disk as XML using the XStream serialization library. Per the framework outlined in Figure 3.1, these serialized runtime objects are then utilized by PANKTI to generate unit tests for the target methods. In total, PANKTI generates 14,222 unit tests from observing the invocations of the 86 methods at runtime, as highlighted in the last row of Table 4.1. Adding these generated tests to the test suite of the three projects, and conducting extreme mutation analysis with Descartes on the updated test suite, we find that 53 of the 86 target methods (61.6%) are no longer pseudo-tested. This means that extreme mutation of these 53 methods is detected by the explicit oracles generated for them by PANKTI.

We illustrate with the example of the target method `getLeftSideBearing`, defined in the class `HorizontalMetricsTable` of PDFBox, presented on lines 1 to 12 of Listing 4.1. This method is covered by five developer-written tests, but is pseudo-tested by the test suite of PDFBox. Consequently, PANKTI identifies `getLeftSideBearing` as a target, and instruments it in order to capture objects associated with it, when it gets invoked at runtime. Using each captured *object profile*, i.e., a unique combination of receiving, argument, and returned objects observed in the field, PANKTI generates 360 unit tests for `getLeftSideBearing`. Figure 4.1 showcases one of these object profiles captured by PANKTI at runtime, which includes a snapshot of the receiving object of type `HorizontalMetricsTable`, the `int` argument `49`, as well as the returned `int` value, `152`. Lines 14 to 25 of Listing 4.1 present the test generated from this object profile of `getLeftSideBearing`. On line 17, the `HorizontalMetricsTable` receiving object is reconstructed from its captured state, by reading a resource file and deserializing its contents. Next, line 18 initializes the `int` argument to `49`. The receiving object and the argument constitute the test input. On line 21, `getLeftSideBearing` is invoked on the receiving object `hMTable`, with the same `gid` as it was called with in the field. Finally, using the explicit oracle on line 24, the equality of the `int` returned from this invocation of the target method, and the one captured at runtime, i.e., 152, is verified. We add the 360 tests generated by PANKTI for `getLeftSideBearing` to the test suite of PDFBox. All of these tests pass, and Descartes no longer reports this target method as being pseudo-tested, owing to the assertion statement within each generated test that clearly specifies the expected behavior of `getLeftSideBearing`.

Per Table 4.1, of the 14,222 tests generated by PANKTI for the three projects, 97.6% pass. The 344 generated tests that fail do so for two key reasons. First,

```
1  public class HorizontalMetricsTable {
2    short[] leftSideBearing;
3    int numHMetrics;
4    ...
5    public int getLeftSideBearing(int gid) {
6      if (gid < numHMetrics)
7        return leftSideBearing[gid];
8      else
9        return nonHorizontalLeftSideBearing[gid - numHMetrics];
10   }
11   ...
12 }
13 ..........................................................................................
14 @Test
15 public void testGetLeftSideBearing {
16   // Arrange
17   HorizontalMetricsTable hMTable = deserialize("getLeftSideBearing-receiving.xml");
18   int gid = 49;

20   // Act
21   int leftSideBearing = hMTable.getLeftSideBearing(gid);

23   // Assert
24   assertEquals(152, leftSideBearing);
25 }
```

Listing 4.1: **(RQ1)** Target method `getLeftSideBearing` defined in the class `HorizontalMetricsTable` of PDFBox is pseudo-tested. Observing one of its invocations in the field, PANKTI generates a unit test with an explicit oracle, which makes `getLeftSideBearing` well-tested.

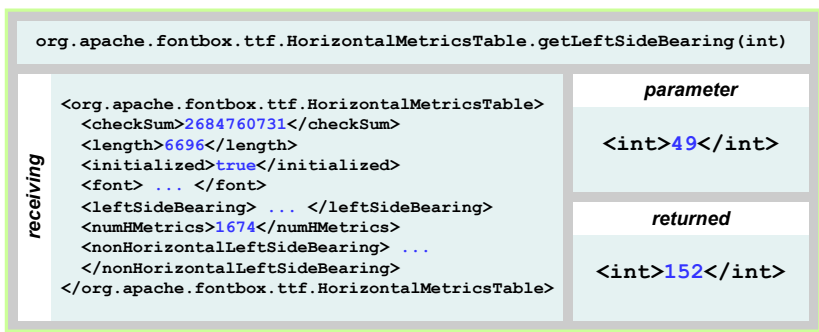


Figure 4.1: An object profile captured by PANKTI for one invocation of the target method `getLeftSideBearing` (lines 5 to 10 of Listing 4.1) in the field.

PANKTI does not override the `equals` method for arbitrary objects, which causes object comparison within the generated assertion to fail. Second, deserialization from captured object states is not guaranteed to reconstruct the exact object observed in the field, especially with regards to external resources and transient fields. Furthermore, 33 of the 86 target methods remain pseudo-tested, despite passing tests generated by PANKTI. We attribute this non-improvement in the test quality of these target methods to mutant equivalence. For example, extreme mutation of a target method that returns a non-primitive object implies replacing the body of the method with a `return null` statement. If the method always returns `null` in the field, this mutation would go undetected by the tests generated by PANKTI.

To what extent do oracles derived from runtime observations contribute to improved test quality?

We devise a mechanism called PANKTI, which observes the invocations of target methods, recording their receiving and argument objects, as well as the object returned from them. PANKTI uses this captured data to generate unit tests for target methods that reflect their behaviors observed in the field. Our results demonstrate that the explicit oracles contained in the generated tests contribute to the improvement in test quality for a majority of the target methods. Extreme mutation of these methods, by removing their body, is no longer undetected, thanks to the oracles generated by PANKTI. We conclude that runtime observations can be used to generate oracles that augment the test quality of the system under test.

4.2 RQ2: To what extent do mock-based oracles derived from runtime observations contribute to regression testing?

We initialize our framework from Section 3.1 into a mechanism called RICK, which observes invocations of target methods as well as the mockable methods within them. The unit tests generated by RICK contain mocks, stubs, and mock-based oracles, that correspond to the data captured in the field.

Following the protocol described in Section 3.3, we evaluate RICK against GEPHI, GRAPHHOPPER, and PDFBOX to answer our second research question. Table 4.2 summarizes the results of this evaluation. In total, RICK targets 128 methods across the three projects, observing one invocation of each method, as well as the invocations of its associated mockable method calls, to generate 294 unit tests. Let us consider the case of GRAPHHOPPER, which has 23 target methods that are triggered in the field. RICK observes one invocation of each of these methods to generate 62 unit tests. Within these tests, 31 objects of external types are mocked, on which 45 methods are called. Moreover, based on

Table 4.2: Summary of results for RQ2: RICK selects methods from three PROJECTS as TARGETS for the generation of unit tests. The number of statements corresponding to the three kinds of mock-based oracles in the generated tests are represented by the columns OO, PO, and CO. A majority of the generated tests (52.4%) SUCCESSFULLY MIMIC, end-to-end, the observed runtime behaviors.

PROJECT	TARGETS	MOCK OBJECTS	MOCK METHODS	STUBS	STATEMENTS			TEST RESULTS		
					OO	PO	CO	UNHANDLED BEHAVIOR	INCOMPLETELY MIMIC	SUCCESSFULLY MIMIC
GEPHI	57	67	93	103	12	94	135	56 / 126	26 / 126	**44 / 126**
GRAPHHOPPER	23	31	45	81	16	88	78	5 / 62	20 / 62	**37 / 62**
PDFBOX	48	53	66	38	10	75	80	22 / 106	11 / 106	**73 / 106**
TOTAL	128	151	204	222	38	257	293	83 / 294 (28.2%)	57 / 294 (19.4%)	**154 / 294 (52.4%)**

observations made at runtime, the mock objects are stubbed using their captured arguments and returned values. There are 81 stubs across the 62 generated tests. Recall from Section 3.3 that RICK generates three kinds of mock-based oracles. The output oracle (OO) verifies the expected output from the target method, the parameter oracle (PO) verifies the arguments passed to the mock methods, while the call oracle (CO) verifies the sequence and frequency of these mock method calls. The OO is expressed as a JUnit5 assertion statement, while the PO and CO are specified through the verification API of Mockito [133, 134]. As highlighted in Table 4.2, the 62 tests generated by RICK for GRAPHHOPPER contain 16 OO statements, 88 PO statements, and 78 CO statements.

We illustrate with the example of Listing 4.2, which presents an excerpt of the class `LineIntIndex` from GRAPHHOPPER. This class declares the field `dataAccess` of type `DataAccess` on line 2, and includes, from lines 4 to 14, the target method `loadExisting`. Within `loadExisting`, there are four calls to two distinct mockable methods on `dataAcess`. The mockable method `loadExisting` is called on line 6, which is followed by three calls to the mockable method `getHeader` with different `int` arguments (lines 9 to 11). RICK observes one invocation of the target method `loadExisting` in the field, and captures its `LineIntIndex` receiving object, as well as the `boolean` value returned value from it. Additionally, within this invocation, RICK also captures the data corresponding to the mockable call to `loadExisting` (i.e., the returned `boolean` value) and the three calls to `getHeader` (i.e., their argument and returned `int`s).

Offline, RICK processes the data captured in the field in order to generate tests. Specifically, the data corresponding to one invocation of a target method is transformed into three unit tests, one for each kind of mock-based oracle. The

```
1  public class LineIntIndex {
2    DataAccess dataAccess;
3    ...
4    public boolean loadExisting() {
5      ...
6      if (!dataAccess.loadExisting())
7        return false;
8      ...
9      GHUtility.checkDAVersion(..., dataAccess.getHeader(0));
10     checksum = dataAccess.getHeader(1 * 4);
11     minResolutionInMeter = dataAccess.getHeader(2 * 4);
12     ...
13     return true;
14   }
15 }
```

Listing 4.2: **(RQ2)** Target method `loadExisting` defined in the class `LineIntIndex` of GraphHopper has mockable method calls on the `dataAccess` field, `loadExisting` on line 6 and `getHeader` on lines 9 to 11.

generated tests include the inputs in the form of the deserialized and receiving argument objects, as well as the mocks and stubs for the external objects. For the target method `loadExisting` of Listing 4.2, RICK generates three tests using the data captured from its invocation. Each test contains one of the mock-based oracles OO, PO, and CO. The common *Arrange* and *Act* phases of these tests are presented in Listing 4.3. As part of the Arrange phase, the receiving `LineIntIndex` object is first deserialized and reconstructed to its observed production state (line 4). On line 5, `DataAccess` is mocked into the `mockDataAccess` object, and inserted as a mocked field within `lineIntIndex`. Next, based on their observed invocations, `loadExisting` and `getHeader` are stubbed, i.e., they are configured to return the observed values given the captured arguments, if any (lines 6 to 9). In the Act phase, the target `loadExisting` method is invoked on `lineIntIndex`. The Assert phase in the test with the output oracle (line 22) verifies that the returned `boolean` value from `loadExisting` is `true`, through the JUnit5 `assertTrue` method. Next, the test with the parameter oracle uses the `verify` API of Mockito (lines 29 to 32) to confirm the arguments with which the mock method calls to `loadExisting` and `getHeader` occur, when invoked on `mockDataAccess`. Finally, the test with the call oracle uses Mockito's `inOrder` to verify that `loadExisting` is first called on `mockDataAcess` once (line 40), followed by three invocations of `getHeader` (line 41). Running the three tests, we find that the OO, PO, and CO all hold. This implies that RICK has successfully transformed runtime observations from the invocation of `loadExisting` into mocks, stubs, and assertion and verification statements.

Overall, as summarized in Table 4.2, 52.4% of the tests generated by RICK pass, demonstrating its capability to generate tests that Successfully Mimic observed runtime behaviors through mocks. With respect to the Incompletely

```
@Test
public void testForloadExisting_XO {
  // Arrange
  LineIntIndex lineIntIndex = deserialize("loadExisting-receiving.xml");
  DataAccess mockDataAccess = mockField_dataAccess_InLineIntIndex(lineIntIndex);
  when(mockDataAccess.loadExisting()).thenReturn(true);
  when(mockDataAccess.getHeader(0)).thenReturn(5);
  when(mockDataAccess.getHeader(4)).thenReturn(1813699);
  when(mockDataAccess.getHeader(8)).thenReturn(300);

  // Act
  boolean actual = lineIntIndex.loadExisting();

  // Assert
  ...
}
.....................................................................................
@Test
public void testForloadExisting_OO {
  ...
  // Assert
  assertTrue(actual);
}
.....................................................................................
@Test
public void testForloadExisting_PO {
  ...
  // Assert
  verify(mockDataAccess, atLeastOnce()).loadExisting();
  verify(mockDataAccess, atLeastOnce()).getHeader(0);
  verify(mockDataAccess, atLeastOnce()).getHeader(4);
  verify(mockDataAccess, atLeastOnce()).getHeader(8);
}
.....................................................................................
@Test
public void testForloadExisting_CO {
  ...
  // Assert
  InOrder orderVerifier = inOrder(mockDataAccess);
  orderVerifier.verify(mockDataAccess, times(1)).loadExisting();
  orderVerifier.verify(mockDataAccess, times(3)).getHeader(anyInt());
}
```

Listing 4.3: **(RQ2)** RICK generates three tests for the target method `loadExisting` of GRAPHHOPPER, each verifying its observed behaviors through one mock-based oracle.

MIMIC and UNHANDLED BEHAVIOR cases of Table 4.2, we find that the generated tests that are not successful at recreating the observed runtime behaviors. We attribute these test failures to incomplete (de)serialization, and side-effects of mocking. We elaborate on these cases in Paper IV.

```
// (Extreme) mutant #1: Lines 5 to 13
public boolean loadExisting() {
  ...
  return true;
  return true;
}

// (Extreme) mutant #2: Lines 5 to 13
public boolean loadExisting() {
  ...
  return true;
  return false;
}

// Mutant #3: Line 6
if (!dataAccess.loadExisting())
if (dataAccess.loadExisting())

// Mutant #4: Line 10
checksum = dataAccess.getHeader(1 * 4);
checksum = dataAccess.getHeader(1 / 4);

// Mutant #5: Line 11
minResolutionInMeter = dataAccess.getHeader(2 * 4);
minResolutionInMeter = dataAccess.getHeader(2 / 4);
```

Listing 4.4: **(RQ2)** Five first-order mutants are introduced in the target method `loadExisting`. The line numbers correspond to the line numbers in Listing 4.2.

We conduct mutation analysis in order to assess the effectiveness of the successful mock-based oracles generated by RICK at detecting regressions. We rely on LittleDarwin [135] to introduce first-order mutants in each target method, before re-running the generated tests to determine if the OO, PO, and CO detect the regression. The five mutants produced for the target method `loadExisting` of Listing 4.2 are presented in Listing 4.4. The OO test generated by RICK detects 2 of these mutants (#2 and #3), as the assertion on line 22 of Listing 4.3 fails on an output that differs from the expected `boolean` value. On the other hand, the CO of lines 39 to 41 kills 3 mutants (#1, #2, and #3), as the invocations to the methods `loadExisting` and `getHeader` are expected on the mocked `DataAccess` object, but do not occur due to the mutation. The verification statements in the PO (lines 29-32) kill all 5 mutants. This is because the expected mock method invocations within the MUT either do not occur entirely, or occur with unexpected arguments, causing the PO test to fail. Overall, we introduce 449 mutants in the target methods in GEPHI, GRAPHHOPPER, and PDFBOX. The mock-based oracles generated by RICK kill 210 of these mutants, and as we see from the example of `loadExisting`, the OO, PO, and CO complement in each other in their ability to detect regressions.

To what extent do mock-based oracles derived from runtime observations contribute to regression testing?

We propose RICK, which observes the invocation of target methods and their mockable method calls. Using the data captured from these observations in the field, RICK generates unit tests for each target method, which mock external objects, and configure their interactions through stubs. The generated tests express the observed behaviors through mock-based oracles, verifying the output from the target method (OO), the arguments that the mock methods should be invoked with (PO), as well as the order in which, and the number of times that, these mock method calls should occur (CO). The ability to generate unit tests with mocks, stubs, and mock-based oracles from runtime observations is novel and unique to RICK. Furthermore, through mutation analysis, we demonstrate that the generated mock-based oracles are good at detecting regressions, and complement each other in their ability to detect bugs.

4.3 RQ3: To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?

Following the protocol of Section 3.3, we use PROZE to capture the arguments of target methods that are directly invoked by developer-written tests. PROZE transforms these tests into parameterized unit tests (PUTs), using the captured arguments to generate argument providers for these PUTs.

Table 4.3 summarizes the results of our evaluation of PROZE on three modules of PDFBox (`fontbox`, `xmpbox`, and `pdfbox`), and two modules of SAML (`query` and `sso`). In total, PROZE identifies 128 target methods in these five modules, which are methods that are directly invoked by the 248 *conventional* unit tests (CUTs) in their developer-written test suites. As detailed in Section 3.3, PROZE isolates each assertion statement contained in these CUTs into its own PUT, which is parameterized over a single target method. Furthermore, each PUT is supplied arguments through a corresponding argument provider, which is generated using the union of arguments captured for the target method from its test and field invocations. In total, PROZE derives 2,287 PUTs from the 248 CUTs. However, we disregard 373 invalid PUTs that fail when supplied with the original arguments of the target method they are parameterized over. As we see from Table 4.3, this results in 1,914 PUTs that are eligible for classification as STRONGLY-COUPLED, FALSIFIABLY-COUPLED, or DECOUPLED. Running these 1,914 PUTs, we find that 686 are strongly-coupled, i.e., their oracle holds only when the target method is invoked with the original argument within the PUT. Next, for 217 PUTs, we discover a valid and invalid set of inputs that make the oracle pass and fail, respectively. These are the falsifiably-coupled cases, where PROZE has successfully

Table 4.3: Summary of results for RQ3: PROZE transforms developer-written conventional unit tests (CUTs) into parameterized unit tests (PUTs). The generated PUTs are classified as STRONGLY-COUPLED, DECOUPLED, or FALSIFIABLY-COUPLED, based on the generalizability of their oracle. PROZE finds 217 falsifiably-coupled oracles, for which it has captured a valid and invalid set of inputs at runtime.

MODULE	TARGET METHODS	CUTS	PUTS	STRONGLY -COUPLED	DECOUPLED	FALSIFIABLY -COUPLED
`fontbox`	12	15	135	31	92	12
`xmpbox`	27	36	543	94	409	40
`pdfbox`	55	155	1,150	510	479	161
`query`	2	3	7	0	7	0
`sso`	32	39	79	51	24	4
TOTAL	128	248	**1,914**	686	1,011	**217**

found generalizable developer-written assertions. Finally, the oracle within 1,011 generated PUTs are valid for all the inputs that are supplied by the argument provider. We refer to this class of PUTs as decoupled.

We illustrate with an end-to-end example from the `xmpbox` module of PDFBOX, whose test suite contains the developer-written CUT `testBagManagement` presented from lines 1 to 17 of Listing 4.5. This CUT calls the target method `createText` on line 8, with a set of four `String` arguments, `null, "rdf", "li", "valueOne"`. The CUT contains four assertions on lines 11, 12, 15 and 16. PROZE derives four PUTs from this CUT, each containing exactly one assertion statement from the CUT (line 27). Furthermore, each PUT is parameterized over `createText`, i.e., it accepts four `String` arguments (line 23) which are then used to invoke `createText` (line 25). The argument provider for these PUTs (lines 31-39) is generated using the 738 unique sets of arguments captured by PROZE for `createText`, while we run the test suite of `xmpbox`, as well as PDFBOX with a field workload. This union contains the original set of arguments (line 34) with which `createText` is invoked with, within `testBagManagement`. Note that each PUT is linked to the generated argument provider through the JUnit5 `@MethodSource` annotation (line 22). Running the generated PUTs, we find that two of them that contain the assertions from lines 11 and 15 pass only when supplied the original argument of `createText`. This makes these two assertions strongly-coupled to the invocation of the target method. Next, the PUT which contains the assertion on line 12 passes for all 738 arguments that are supplied by the provider. We therefore conclude that this oracle is decoupled from the invocation of `createText`. Finally, the PUT with the assertion on line 16 holds for the original set of arguments (line 34), as well as exactly one additional set,

```java
@Test
public void testBagManagement() {
  XMPMetadata parent = XMPMetadata.createXMPMetadata();
  XMPSchema schem = new XMPSchema(parent, "nsURI", "nsSchem");
  String bagName = "BAGTEST";
  String value1 = "valueOne";
  String value2 = "valueTwo";
  schem.addBagValue(bagName, schem.getMetadata().getTypeMapping().createText(null,
      "rdf", "li", value1));
  schem.addQualifiedBagValue(bagName, value2);
  List<String> values = schem.getUnqualifiedBagValueList(bagName);
  assertEquals(value1, values.get(0));
  assertEquals(value2, values.get(1));
  schem.removeUnqualifiedBagValue(bagName, value1);
  List<String> values2 = schem.getUnqualifiedBagValueList(bagName);
  assertEquals(1, values2.size());
  assertEquals(value2, values2.get(0));
}
.....................................................................................

// Generated PUT parameterized over createText(String, String, String, String)
@ParameterizedTest
@MethodSource("provideCreateTextArgs")
public void testBagManagement_PUT_N(String ns, String prefix, String propName,
    String value) {
  ...
  schem.addBagValue(bagName, schem.getMetadata().getTypeMapping().createText(ns,
      prefix, propName, value));
  ...
  // one of lines 11, 12, 15, or 16 per PUT
  ...
}

// Argument provider with 738 captured inputs
static Stream<Arguments> provideCreateTextArgs() {
  return java.util.stream.Stream.of(
    Arguments.of(null, "rdf", "li", "valueOne"), // original argument
    ...
    Arguments.of(null, "pdf", "PDFVersion", "1.4"),
    Arguments.of("nsURI", "nsSchem", "li", "valueTwo")
  );
}
```

Listing 4.5: **(RQ3)** The CUT `testBagManagement` in `xmpbox` has 4 assertions, and directly calls target method `createText`. PROZE captures 738 arguments for `createText` at runtime, and generates the argument provider method `provideCreateTextArgs`. This provider supplies arguments to 4 generated PUTs parameterized over `createText`. The parts common to the 4 generated PUTs are included in `testBagManagement_PUT_N`.

i.e., `"nsURI", "nsSchem", "li", "valueTwo"` (line 37). We classify this as a falsifiably-coupled case.

As we see from Table 4.3, PROZE successfully identifies 217 falsifiably-coupled developer-written oracles that are provably sensitive to valid and invalid test inputs captured at runtime, and that are generalizable to a larger input range. We envision that these PUTs can be integrated into the test suite after (i) limiting the argument provider to deliver only valid arguments, by removing the falsifying arguments from the initial provider; and (ii) if multiple PUTs use the same argument providers, merging the assertions of these different PUTs into one PUT.

To what extent do runtime observations contribute to the generalization of existing, developer-written oracles?

PROZE is a novel approach for the transformation of developer-written unit tests into parameterized unit tests (PUTs). It captures the union of arguments passed to a target method at runtime to generate argument providers that supply values to each generated PUT, parameterized over a single target method and containing a single developer-written oracle. With PROZE, we demonstrate that developers indeed write oracles that generalize over a larger input range than the one they envision. PROZE successfully harnesses runtime observations to discover such oracles in existing test suites.

4.4 Conclusion about observation-based oracle generation

Each of our research questions highlights a distinct way of expressing observed runtime behaviors as test oracles. First, the explicit assertions generated by PANKTI (RQ1) guarantee that regressions, including those introduced by extreme mutations, do not go undetected by the test suite. The generated tests therefore improve the test quality of the target project based on the defined test adequacy criterion of mutation analysis. Second, the mock-based oracles generated by RICK (RQ2) demonstrate that different aspects of field behavior can be expressed using the configured machinery of mocks and stubs. These mock-based oracles are capable of detecting regressions that are introduced in the system as it evolves. Third, with PROZE (RQ3) we discover existing developer-written oracles that are valid over a larger range of inputs captured at runtime, allowing the test suite to be strengthened through parameterized unit tests. Runtime observations can indeed contribute to the generalization of developer-written oracles. These findings are novel to the best of our knowledge. They address the diverse ways in which runtime observations can be incorporated into the test suite to augment it, such that the nominal operative behaviors of the target system are better represented. Our approach can be applied to software projects regardless of the maturity of their current testing practices. The oracles produced using runtime observations act not only as feedback from real behaviors into the test suite, but

also strengthen the test suite by facilitating the detection of regressions as the system inevitably evolves with time.

Per the survey by Bertolino *et al.* [18] on field-based testing strategies, we position our contributions as ex-vivo techniques that use data from the field to generate tests in the development environment. The generated tests are intended to complement the developer-written test suite, and be run in-house. Furthermore, with respect to the literature on automated generation of oracles summarized in Section 2.1, our work is classified under behavior-based oracle generation strategies. The novelty of our work lies in the utilization of real-world behaviors for the generation of regression oracles (RQ1, RQ2) and mock-based oracles (RQ2), as well as for the generalization of existing developer-written oracles (RQ3). Through our contributions, we propose a solution to the oracle problem [12] by observing systems in the field to determine their expected behaviors. Moreover, we generate complete tests by instantiating our framework of Section 3.1. Relative to the observation-based test generation strategies discussed in Section 2.2, each test we generate includes both of the components essential to a software test, its inputs and its oracle, derived from runtime behaviors.

We note here that, as with related work on observation-based test generation [18,112], our approach relies on monitoring and serialization at runtime. There are performance implications of this test generation strategy that must be considered. With PANKTI, RICK, and PROZE, we instrument target methods to serialize data related to their invocations. For PROZE, our evaluation focuses on target methods with primitive or `String` arguments, for which the serialization cost is minimal. However, with PANKTI and RICK, that also serialize non-primitive objects, we employ strategies to limit the cost incurred from monitoring the invocations of target methods. First, we define criteria to select target methods that are useful for test generation, such as targeting pseudo-tested methods with PANKTI, or methods with mockable method calls with RICK. Second, during the execution of the test suite, or the project in the field, a target method may be invoked thousands of times. We limit the number of invocations, and the size of the captured data on disk, to sample a subset of these invocations. We discuss the performance implications of PANKTI and RICK in Papers I and IV, respectively.

5. Conclusion

"All these sights and sounds and smells will be yours to enjoy, Wilbur — this lovely world, these precious days..."

E.B. White, Charlotte's Web

The process of software testing is key to ensuring the quality of all the diverse software systems that we have come to depend on. In this thesis, we have provided an overview of the vibrant ecosystem of software testing techniques, especially focusing on strategies for the automated generation of software tests. Our own contributions in this domain are motivated by the observation that, despite significant developer effort, it is seldom possible to account for, and test, all the scenarios that a software system might encounter in the field [18]. In fact, the oracles contained within developer-written tests might not reflect behaviors that are exhibited by a system in production, when it is exercised with real workloads by end-users [16]. This gap between tested behaviors and field behaviors is not addressed explicitly by the techniques proposed in the literature on automated software test generation.

5.1 Summary

We provide evidence that runtime observations can be harnessed for the generation of test inputs and test oracles. In order to do so, we propose a novel conceptual framework for observing a system as it runs in the field, capturing runtime behaviors in the form of data, and using this data to generate tests. The generated tests, including test inputs and oracles, are representative of the observed field behaviors, and are intended to complement developers in their testing activities. Furthermore, we provide three distinct instances of our conceptual framework, targeting diverse ways of utilizing runtime observations for the augmentation of test oracles.

First, with PANKTI, we target methods that are inadequately tested by the developer-written test suite of the system under test. PANKTI observes the invocations of these methods in the field, and captures the objects associated with the invocations. It then uses these captured objects to generate unit tests that

recreate the observed production state. Through our evaluation with pseudo-tested methods within three notable Java projects, we demonstrate that over 60% of them are no longer pseudo-tested as a consequence of the explicit oracles contained within the tests generated by PANKTI.

Second, we propose a mechanism called RICK that captures objects related to the invocations of target methods, as well as the methods within it. These objects are used by RICK to generate unit tests with mocks, stubs, and three kinds of mock-based oracles. The output oracle verifies that the output from the target method within the test is the same as the output observed in the field. The parameter oracle verifies the arguments with which nested method calls occur within the invocation of the target method, and the call oracle verifies the expected sequence and frequency of these nested method calls. These three mock-based oracles also detect regressions introduced within the system. A majority of the tests generated by RICK for three large Java projects successfully reproduce the entirety of the observed field behaviors.

Third, PROZE targets methods that are directly invoked by developer-written unit tests. It captures the arguments with which these target methods are invoked in the field and during the execution of the test suite, and generates data providers with these runtime arguments. The data providers supply arguments to parameterized unit tests that are derived from the existing unit tests. From our evaluation of PROZE against five Java modules from two open-source projects, we discover over 200 existing developer-written oracles that are provably generalizable. This means that, for these oracles, we find both valid and invalid inputs, and demonstrate that they indeed hold over a larger set of input values than expressed in the original unit tests authored by developers.

Overall, these results provide strong and sound support to our core thesis:

> **Runtime observations, made as a system executes in the field, can be utilized to generate complete tests, with inputs and oracles. The generated tests augment the developer-written test suite, serving as regression tests that are representative of real-world usage scenarios.**

5.2 Future work

We are hopeful that our findings will encourage software developers to incorporate observation-based testing in their practice. Additionally, we are optimistic about the implications of our contributions within software engineering research. In particular, we propose three directions that further our work on harnessing runtime observations for augmenting software quality.

Deriving invariants from runtime observations for use as oracles: Per our methodology, the oracles we generate are fine-tuned to the exact production state recreated within the generated tests. However, as we demonstrate with

PROZE in Paper IV, it is desirable to discover oracles that are less tightly coupled with specific input scenarios. Such oracles can express more general behaviors of the system, and can be likened to invariants [53] or properties [43] that generalize to a large input space. We propose that it is possible to use runtime observations to derive such invariants that are valid for a large set of representative field inputs. In order to do this, target units would be monitored in the field to facilitate the development of a *sense of self* [136] within the system. This implies that there would be an awareness within the system with respect to its nominal functional behaviors, from which invariants can be derived. For example, given enough observations for a method, invariants can be deduced on the basis of a range of inputs and outputs that are valid for the method. Additionally, the range of inputs that result in equivalent outputs, or the states of internal variables [39] can serve as invariants. The invariants may also correspond to metamorphic relations that are derived from the change in the states of production objects. The invariants can be utilized as oracles within generated tests. They may also be embedded in production code, to dynamically ensure correctness and safeguard against deviations from expected behaviors [137].

Deriving oracles by leveraging usage from client projects: In this thesis, we evaluate our approach against executable projects for which we design workloads that resemble usage scenarios by human users. However, most software projects are actually intended to be used as *libraries* or dependencies by a large number of diverse *client* projects [138]. For such libraries, the field workload can be considered as their usage by the client projects, either within their developer-written test suite, or in the field. There are studies that investigate the advantages of adopting client usage information to strengthen the test quality of libraries [139, 140]. We envision that our proposed framework can be adapted to generate tests for library projects based on its usage by diverse clients. Moreover, an interesting application of this technique would be in the context of domain-specific libraries. For example, multiple JSON serialization libraries exist for all programming languages, each following the JSON specification [122]. Oracles can be automatically produced by evaluating the usage of JSON libraries across client projects and programming languages. This empirical analysis will shed light on the behavioral diversity among libraries that implement the same specification. Additionally, we will gain insight into the distinct ways in which their APIs are used within client projects. The oracles generated from this technique may also be transplanted [141] across the libraries.

Runtime data, faking, and generative AI for test inputs: The inputs and oracles contained in the tests generated in this thesis are sourced purely from runtime observations. Meanwhile, data faking is a useful way of supplying realistic inputs within automated tests. Additionally, generative AI models have recently found application in almost all software development tasks. An initial investigation into the use of data fakers with generative AI looks promising [142]. We hypothesize that using runtime data in conjunction with data faking and

generative AI can be fruitful for producing more representative test inputs. One way to achieve this would be to fine-tune a generative model on a sample of captured runtime data. It can then be used to produce an in-house data faker that is capable of supplying interesting and realistic test inputs to tests that are manually created or automatically generated. We believe that this strategy would also be valuable across diverse natural and programming languages.

References

[1] B. Baudry and M. Monperrus, "Exhaustive Survey of Rickrolling in Academic Literature," in *Proceedings of SIGBOVIK'22*, 2022.

[2] C. Soto-Valero, M. Monperrus, and B. Baudry, "The Multibillion Dollar Software Supply Chain of Ethereum," *Computer*, vol. 55, no. 10, pp. 26–34, 2022.

[3] B. Baudry and M. Monperrus, "Programming Art With Drawing Machines," *Computer*, vol. 57, no. 07, pp. 104–108, 2024.

[4] W. E. Wong, X. Li, and P. A. Laplante, "Be More Familiar With Our Enemies and Pave the Way Forward: A Review of the Roles Bugs Played in Software Failures," *Journal of Systems and Software*, vol. 133, pp. 68–94, 2017.

[5] M. Staats, M. W. Whalen, and M. P. Heimdahl, "Programs, Tests, and Oracles: The Foundations of Testing Revisited," in *Proceedings of the 33rd international conference on software engineering*, pp. 391–400, 2011.

[6] M. Aniche, C. Treude, and A. Zaidman, "How Developers Engineer Test Cases: An Observational Study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2021.

[7] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: A Survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[8] L. Gazzola, L. Mariani, F. Pastore, and M. Pezze, "An Exploratory Study of Field Failures," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 67–77, IEEE, 2017.

[9] G. Jahangirova, *Oracle Assessment, Improvement and Placement*. PhD thesis, UCL (University College London), 2019.

[10] J. Leveau, X. Blanc, L. Réveillère, J.-R. Falleri, and R. Rouvoy, "Fostering the Diversity of Exploratory Testing in Web Applications," *Software Testing, Verification and Reliability*, vol. 32, no. 5, p. e1827, 2022.

[11] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, *et al.*, "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation," *Journal of systems and software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[13] W. Sun, Z. Guo, M. Yan, Z. Liu, Y. Lei, and H. Zhang, "Method-Level Test-to-Code Traceability Link Construction by Semantic Correlation Learning," *IEEE Transactions on Software Engineering*, 2024.

[14] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On Learning Meaningful Assert Statements for Unit Test Cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 1398–1409, 2020.

[15] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, IEEE, 2015.

[16] Q. Wang, Y. Brun, and A. Orso, "Behavioral Execution Comparison: Are Tests Representative of Field Behavior?," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 321–332, IEEE, 2017.

[17] Q. Wang and A. Orso, "Mimicking User Behavior to Improve In-House Test Suites," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 318–319, IEEE, 2019.

[18] A. Bertolino, P. Braione, G. D. Angelis, L. Gazzola, F. Kifetew, L. Mariani, M. Orrù, M. Pezze, R. Pietrantuono, S. Russo, *et al.*, "A Survey of Field-Based Testing Techniques," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–39, 2021.

[19] N. Alshahwan, M. Harman, and A. Marginean, "Software Testing Research Challenges: An Industrial Perspective," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, IEEE, 2023.

[20] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 4, pp. 1–49, 2015.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 75–84, IEEE, 2007.

[22] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 416–419, 2011.

[23] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[24] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software Testing With Large Language Models: Survey, Landscape, and Vision," *IEEE Transactions on Software Engineering*, 2024.

[25] N. Li and J. Offutt, "Test Oracle Strategies for Model-based Testing," *IEEE Transactions on Software Engineering*, vol. 43, no. 4, pp. 372–395, 2016.

[26] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè, "Cross-checking Oracles from Intrinsic Software Redundancy," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 931–942, 2014.

[27] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic Generation of Oracles for Exceptional Behaviors," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 213–224, 2016.

[28] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos, "Translating Code Comments to Procedure Specifications," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pp. 242–253, 2018.

[29] M. Motwani and Y. Brun, "Automatically Generating Precise Oracles from Structured Natural Language Specifications," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 188–199, IEEE, 2019.

[30] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[31] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2008.

[32] P. Loyola, M. Staats, I.-Y. Ko, and G. Rothermel, "Dodona: Automated Oracle Data Set Selection," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 193–203, 2014.

[33] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "Automated Oracle Data Selection Support," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1119–1137, 2015.

[34] C. Xu, V. Terragni, H. Zhu, J. Wu, and S.-C. Cheung, "MR-Scout: Automated Synthesis of Metamorphic Relations from Existing Test Cases," *ACM Transactions on Software Engineering and Methodology*, 2024.

[35] F. Pastore, L. Mariani, and G. Fraser, "CrowdOracles: Can the Crowd Solve the Oracle Problem?," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 342–351, IEEE, 2013.

[36] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pp. 263–272, IEEE, 2017.

[37] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A Comprehensive Survey of Trends in Oracles for Software Testing," *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[38] N. Li and J. Offutt, "An Empirical Analysis of Test Oracle Strategies for Model-Based Testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 363–372, IEEE, 2014.

[39] Y. Xiong, D. Hao, L. Zhang, T. Zhu, M. Zhu, and T. Lan, "Inner Oracles: Input-specific Assertions on Internal States," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 902–905, 2015.

[40] M. Nassif, A. Hernandez, A. Sridharan, and M. P. Robillard, "Generating Unit Tests for Documentation," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3268–3279, 2021.

[41] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pp. 815–816, 2007.

[42] D. Saff, M. Boshernitsan, and M. D. Ernst, "Theories in Practice: Easy-To-Write Specifications That Catch Bugs," tech. rep., MIT Computer Science and Artificial Intelligence Laboratory, 2008.

[43] K. Claessen and J. Hughes, "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pp. 268–279, 2000.

[44] D. R. MacIver, Z. Hatfield-Dodds, *et al.*, "Hypothesis: A New Approach to Property-Based Testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[45] H. Goldstein, J. W. Cutler, D. Dickstein, B. C. Pierce, and A. Head, "Property-Based Testing in Practice," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[46] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable Modified Condition/Decision Coverage," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 102–111, IEEE, 2013.

[47] T. Xie, "Augmenting Automatically Generated Unit-Test Suites With Regression Oracle Checking," in *European Conference on Object-Oriented Programming*, pp. 380–403, Springer, 2006.

[48] M. Staats, G. Gay, and M. P. Heimdahl, "Automated Oracle Creation Support, Or: How I Learned to Stop Worrying About Fault Propagation and Love Mutation Testing," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 870–880, IEEE, 2012.

[49] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2013.

[50] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A Survey on Metamorphic Testing," *IEEE Transactions on software engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[51] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic Testing: A Review of Challenges and Opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[52] A. Blasi, A. Gorla, M. D. Ernst, M. Pezzè, and A. Carzaniga, "MeMo: Automatically Identifying Metamorphic Relations in Javadoc Comments for Test Automation," *Journal of Systems and Software*, vol. 181, p. 111041, 2021.

[53] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[54] M. Staats, S. Hong, M. Kim, and G. Rothermel, "Understanding User Understanding: Determining Correctness of Generated Program Invariants," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 188–198, 2012.

[55] C. Csallner, N. Tillmann, and Y. Smaragdakis, "DySy: Dynamic Symbolic Execution for Invariant Inference," in *Proceedings of the 30th international conference on Software engineering*, pp. 281–290, 2008.

[56] H. Yu, Y. Lou, K. Sun, D. Ran, T. Xie, D. Hao, Y. Li, G. Li, and Q. Wang, "Automated Assertion Generation via Information Retrieval and Its Integration With Deep Learning," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 163–174, 2022.

[57] F. Molina, "Applying Learning Techniques to Oracle Synthesis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1153–1157, 2020.

[58] F. Molina, P. Ponzio, N. Aguirre, and M. Frias, "EvoSpex: An Evolutionary Algorithm for Learning Postconditions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1223–1235, IEEE, 2021.

[59] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A Neural Method for Test Oracle Generation," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2130–2141, 2022.

[60] G. Fraser and A. Arcuri, "Evolutionary Generation of Whole Test Suites," in *2011 11th International Conference on Quality Software*, pp. 31–40, IEEE, 2011.

[61] S. B. Hossain, A. Filieri, M. B. Dwyer, S. Elbaum, and W. Visser, "Neural-based Test Oracle Generation: A Large-scale Evaluation and Lessons Learned," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 120–132, 2023.

[62] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards More Realistic Evaluation for Neural Test Oracle Generation," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 589–600, 2023.

[63] J. Shin, H. Hemmati, M. Wei, and S. Wang, "Assessing Evaluation Metrics for Neural Test Oracle Generation," *IEEE Transactions on Software Engineering*, 2024.

[64] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand, "Perfect is the Enemy of Test Oracle," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 70–81, 2022.

[65] F. Molina and A. Gorla, "Test Oracle Automation in the Era of LLMs," *arXiv preprint arXiv:2405.12766*, 2024.

[66] S. B. Hossain and M. Dwyer, "TOGLL: Correct and Strong Test Oracle Generation with LLMs," *arXiv preprint arXiv:2405.03786*, 2024.

[67] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A Snowballing Literature Study on Test Amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019.

[68] M. Staats, P. Loyola, and G. Rothermel, "Oracle-Centric Test Case Prioritization," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pp. 311–320, IEEE, 2012.

[69] D. Schuler and A. Zeller, "Assessing Oracle Quality with Checked Coverage," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 90–99, IEEE, 2011.

[70] D. Schuler and A. Zeller, "Checked Coverage: An Indicator for Oracle Quality," *Software testing, verification and reliability*, vol. 23, no. 7, pp. 531–551, 2013.

[71] G. Fraser and A. Zeller, "Mutation-driven Generation of Unit Tests and Oracles," in *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 147–158, 2010.

[72] R. Niedermayr, E. Juergens, and S. Wagner, "Will My Tests Tell Me if I Break This Code?," in *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*, pp. 23–29, 2016.

[73] O. L. Vera-Pérez, B. Danglot, M. Monperrus, and B. Baudry, "A Comprehensive Study of Pseudo-Tested Methods," *Empirical Software Engineering*, vol. 24, pp. 1195–1225, 2019.

[74] C. Huo and J. Clause, "Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 621–631, 2014.

[75] X. Guo, M. Zhou, X. Song, M. Gu, and J. Sun, "First, Debug the Test Oracle," *IEEE Transactions on Software Engineering*, vol. 41, no. 10, pp. 986–1000, 2015.

[76] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test Oracle Assessment and Improvement," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 247–258, 2016.

[77] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "OASIs: Oracle Assessment and Improvement Tool," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 368–371, 2018.

[78] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An Empirical Validation of Oracle Improvement," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1708–1728, 2019.

[79] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary Improvement of Assertion Oracles," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1178–1189, 2020.

[80] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "GAssert: A Fully Automated Tool to Improve Assertion Oracles," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 85–88, IEEE, 2021.

[81] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, "AGORA: Automated Generation of Test Oracles for REST APIs," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1018–1030, 2023.

[82] S. Tuli, K. Bojarczuk, N. Gucevska, M. Harman, X.-Y. Wang, and G. Wright, "Simulation-Driven Automated End-to-End Test and Oracle Inference," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 122–133, IEEE, 2023.

[83] L. Mariani, M. Pezzè, and D. Zuddas, "Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles," in *Proceedings of the 40th international conference on software engineering*, pp. 280–290, 2018.

[84] A. E. Genç, H. Sözer, M. F. Kıraç, and B. Aktemur, "ADVISOR: An Adjustable Framework for Test Oracle Automation of Visual Output Systems," *IEEE Transactions on Reliability*, vol. 69, no. 3, pp. 1050–1063, 2019.

[85] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, "On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 957–970, 2014.

[86] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 183–192, IEEE, 2014.

[87] K. Baral, J. Johnson, J. Mahmud, S. Salma, M. Fazzini, J. Rubin, J. Offutt, and K. Moran, "Automating GUI-based Test Oracles for Mobile Apps," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 309–321, 2024.

[88] R. Jabbarvand, F. Mehralian, and S. Malek, "Automated Construction of Energy Test Oracles for Android," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 927–938, 2020.

[89] G. Gay, S. Rayadurgam, and M. P. Heimdahl, "Automated Steering of Model-based Test Oracles to Admit Real Program Behaviors," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 531–555, 2016.

[90] A. Afzal, C. Le Goues, and C. S. Timperley, "Mithra: Anomaly Detection as an Oracle for Cyberphysical Systems," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4535–4552, 2021.

[91] A. Arrieta, M. Otaegi, L. Han, G. Sagardui, S. Ali, and M. Arratibel, "Automating Test Oracle Generation in DevOps for Industrial Elevators," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 284–288, IEEE, 2022.

[92] A. Gartziandia, A. Arrieta, J. Ayerdi, M. Illarramendi, A. Agirre, G. Sagardui, and M. Arratibel, "Machine Learning-based Test oracles for Performance Testing of Cyber-physical Systems: An Industrial Case Study on Elevators Dispatching Algorithms," *Journal of Software: Evolution and Process*, vol. 34, no. 11, p. e2465, 2022.

[93] G. Jahangirova, A. Stocco, and P. Tonella, "Quality Metrics and Oracles for Autonomous Vehicles Testing," in *2021 14th IEEE conference on software testing, verification and validation (ICST)*, pp. 194–204, IEEE, 2021.

[94] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 213–223, 2005.

[95] K. Sen and G. Agha, "CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools," in *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*, pp. 419–423, Springer, 2006.

[96] N. Tillmann and J. De Halleux, "Pex–White Box Test Generation for .NET," in *International conference on tests and proofs*, pp. 134–153, Springer, 2008.

[97] S. Liu and S. Nakajima, "Automatic Test Case and Test Oracle Generation Based on Functional Scenarios in Formal Specifications for Conformance Testing," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 691–712, 2020.

[98] X. Xiao, T. Xie, N. Tillmann, and J. De Halleux, "Precise Identification of Problems for Structural Test Generation," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 611–620, 2011.

[99] N. Alshahwan, A. Blasi, K. Bojarczuk, A. Ciancone, N. Gucevska, M. Harman, M. Krolikowski, R. Rojas, D. Martac, S. Schellaert, *et al.*, "Enhancing Testing at Meta with Rich-State Simulated Populations," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pp. 1–12, 2024.

[100] G. Fraser and A. Zeller, "Exploiting Common Object Usage in Test Case Generation," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 80–89, IEEE, 2011.

[101] A. Deljouyi and A. Zaidman, "Generating Understandable Unit Tests Through End-To-End Test Scenario Carving," in *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 107–118, IEEE, 2023.

[102] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A Capture/Replay Tool for Observation-Based Testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 158–167, 2000.

[103] B. Pasternak, S. Tyszberowicz, and A. Yehudai, "GenUTest: A Unit Test and Mock Aspect Generation Tool," *International journal on software tools for technology transfer*, vol. 11, pp. 273–290, 2009.

[104] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving Differential Unit Test Cases From System Test Cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 253–264, 2006.

[105] A. Kampmann and A. Zeller, "Carving Parameterized Unit Tests," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 248–249, IEEE, 2019.

[106] 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*, pp. 159–170, 2010.

[107] D. Qu, C. Zhao, Y. Jiang, and C. Xu, "Towards Life-Long Software Self-Validation in Production," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, pp. 357–366, 2024.

[108] L. Zhang, B. Morin, P. Haller, B. Baudry, and M. Monperrus, "A Chaos Engineering System for Live Analysis and Falsification of Exception-Handling in the JVM," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2534–2548, 2019.

[109] M. Ceccato, D. Corradini, L. Gazzola, F. M. Kifetew, L. Mariani, M. Orru, and P. Tonella, "A Framework for In-Vivo Testing of Mobile Applications," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 286–296, IEEE, 2020.

[110] L. Gazzola, L. Mariani, M. Orrú, M. Pezze, and M. Tappler, "Testing Software in Production Environments With Data From the Field," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 58–69, IEEE, 2022.

[111] A. Bertolino, G. De Angelis, B. Miranda, and P. Tonella, "In Vivo Test and Rollback of Java Applications as They Are," *Software Testing, Verification and Reliability*, vol. 33, no. 7, p. e1857, 2023.

[112] N. Alshahwan, M. Harman, A. Marginean, R. Tal, and E. Wang, "Observation-based Unit Test Generation at Meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pp. 173–184, 2024.

[113] S. Messaoudi, D. Shin, A. Panichella, D. Bianculli, and L. C. Briand, "Log-Based Slicing for System-Level Test Cases," in *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pp. 517–528, 2021.

[114] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic Generation of System Test Cases from Use Case Specifications," in *Proceedings of the 2015 international symposium on software testing and analysis*, pp. 385–396, 2015.

[115] B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry, "An approach and benchmark to detect behavioral changes of commits in continuous integration," *Empirical Software Engineering*, vol. 25, pp. 2379–2415, 2020.

[116] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing Non-Adequate Test Suites Using Coverage Criteria," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pp. 302–313, 2013.

[117] M. Hilton, J. Bell, and D. Marinov, "A Large-Scale Study of Test Coverage Evolution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 53–63, 2018.

[118] G. Petrović, M. Ivanković, G. Fraser, and R. Just, "Practical Mutation Testing at Scale: A View from Google," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3900–3912, 2021.

[119] K. Jain, G. T. Kalburgi, C. Le Goues, and A. Groce, "Mind the Gap: The Difference Between Coverage and Mutation Score Can Guide Testing Efforts," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 102–113, 2023.

[120] Q. Wang and A. Orso, "Improving testing by mimicking user behavior," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 488–498, IEEE, 2020.

[121] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pp. 177–182, IEEE, 2012.

[122] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "The Behavioral Diversity of Java JSON Libraries," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 412–422, IEEE, 2021.

[123] N. Tillmann and W. Schulte, "Parameterized unit tests," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 253–262, 2005.

[124] C. Wei, L. Xiao, T. Yu, X. Chen, X. Wang, S. Wong, and A. Clune, "Automatically Tagging the "AAA" Pattern in Unit Test Cases Using Machine Learning Models," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–3, 2022.

[125] T. Mackinnon, S. Freeman, and P. Craig, "Endo-testing: unit testing with mock objects," *Extreme programming examined*, pp. 287–301, 2000.

[126] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[127] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "To mock or not to mock? an empirical study on mocking practices," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 402–412, IEEE, 2017.

[128] H. Zhu, L. Wei, V. Terragni, Y. Liu, S.-C. Cheung, J. Wu, Q. Sheng, B. Zhang, and L. Song, "Stubcoder: Automated generation and repair of stub code for mock objects," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 1, pp. 1–31, 2023.

[129] H. Du, V. K. Palepu, and J. A. Jones, "Ripples of a mutation—an empirical study of propagation effects in mutation testing," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13, 2024.

[130] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proceedings of the 2011 international symposium on software testing and analysis*, pp. 364–374, 2011.

[131] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The maven dependency graph: a temporal graph-based representation of maven central," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 344–348, IEEE, 2019.

[132] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[133] S. Mostafa and X. Wang, "An Empirical Study on the Usage of Mocking Frameworks in Software Testing," in *2014 14th international conference on quality software*, pp. 127–132, IEEE, 2014.

[134] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli, "Mock Objects for Testing Java Systems: Why and How Developers Use Them, and How They Evolve," *Empirical Software Engineering*, vol. 24, pp. 1461–1498, 2019.

[135] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: a feature-rich and extensible mutation testing framework for large and complex java systems," in *Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers 7*, pp. 148–163, Springer, 2017.

[136] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," in *Proceedings 1996 IEEE symposium on security and privacy*, pp. 120–128, IEEE, 1996.

[137] A. Sharma, M. Wittlinger, B. Baudry, and M. Monperrus, "SBOM. EXE: Countering Dynamic Code Injection based on Software Bill of Materials in Java," *arXiv preprint arXiv:2407.00246*, 2024.

[138] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies," *IEEE Transactions on Software Engineering*, 2023.

[139] I. Schittekat, M. Abdi, and S. Demeyer, "Can We Increase the Test-Coverage in Libraries using Dependent Projects' Test-Suites?," in *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, pp. 294–298, 2022.

[140] M. Abdi and S. Demeyer, "Test Transplantation Through Dynamic Test Slicing," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 35–39, IEEE, 2022.

[141] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated Software Transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 257–269, 2015.

[142] B. Baudry, K. Etemadi, S. Fang, Y. Gamage, Y. Liu, Y. Liu, M. Monperrus, J. Ron, A. Silva, and D. Tiwari, "Generative AI to Generate Test Data Generators," *IEEE Software*, 2024.

Part B