# Leveraging the External Resources and Meta-data to Highlight the Gap between a Program's Implementation and its Documentation

By

Devika Sondhi

Under the Supervision of

Prof. Rahul Purandare, IIIT-Delhi.

Indraprastha Institute of Information Technology, Delhi

# Leveraging the External Resources and Meta-data to Highlight the Gap between a Program's Implementation and its Documentation

By

Devika Sondhi

Submitted

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

to the

Indraprastha Institute of Information Technology, Delhi

27 June, 2021

# Certificate

This is to certify that the thesis titled - **"Leveraging the External Resources and Meta-data to Highlight the Gap between a Program's Implementation and its Documentation"** being submitted by **Devika Sondhi** to Indraprastha Institute of Information Technology, Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree/diploma.

Prof. Rahul Purandare

27 June, 2021
Department of Computer Science, IIIT-Delhi

# Acknowledgments

I am deeply thankful to my advisor, Dr. Rahul Purandare, for his guidance and support. I attribute a major part of my interest in the field of research to him. His knowledge and the efforts to continuously expand it, the quality of striving for high quality and work ethics shall continue to inspire me. Some of the key learnings during my venture with him has been to be able to critique a work and take criticism as an opportunity to improve. These learnings are invaluable and shall certainly help me in my research career and otherwise.

This dissertation has been supported under the Prime Minister's Fellowship Scheme for Doctoral Research, a PPP initiative of Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India and Confederation of Indian Industry (CII). I thank the Apex Council and all those associated with this initiative. Thanks to Ms. Neha Gupta, Mr. Ravi Hira and Ms. Shalini Verma for the efficient management of the fellowship and for being available to address my queries. This fellowship has been jointly supported by Microsoft Research, as the industry partner. I owe my gratitude to Dr. Kapil Vaswani, my industry mentor, and Microsoft Research for kindly offering the fellowship support and other opportunities.

I'm grateful to my thesis examiners, Dr. Serge Demeyer, from the University of Antwerp, Dr. Tevfik Bultan, from the University of California at Santa Barbara, Dr. Andy Zaidman, from Delft University of Technology. Their time and constructive feedback have been important in shaping this thesis. I'm also fortunate to have obtained regular feedback on my progress from the annual evaluation committee members, Dr. Piyus Kedia and Dr. Vivek Kumar, and the comprehensive examination committee members, Dr. Amey Karkare, Dr. Saket Anand and Dr. Sambuddho Chakravarty.

I sincerely thank the faculty at IIIT Delhi for providing me the knowledge and the research-oriented training through the course-work. A special thanks to Dr. Pushpendra Singh for mentoring me during my early experience in research. This experience has been one of the factors behind my choice to pursue a Ph.D.

I would like to express my gratitude towards the administrative staff, the IT-team and the support staff at IIIT Delhi. Their best efforts and help have allowed me to focus on my research-related activities without any significant distraction.

Thanks to my lab-mates, Venkatesh Vinayakarao, Dhriti Khanna, Ridhi Jain, Nikita Mehrotra and Khushboo Chitre. It has been a memorable time spent with each one of you, be it at a casual chat over meals, brainstorming ideas, sharing feedback on each other's work, celebrating each other's achievements or lending an ear during the low phases of this journey and helping each other come out of it.

It has been an enriching experience working with my collaborators, Venkatesh Vinayakarao, Divya Rani, Mayank Jobanputra, Sakshi Sharma, Salil Purandare, Avyakt Gupta, Ankit Kumar and Deepanshu Kaushal. I'm deeply thankful to them.

This journey would not have been as rewarding as it is without the unconditional love and support of my parents, Sanjeev and Vini, and my sibling, Anirudh. They have been very understanding and encouraging towards my research commitments and that has been extremely valuable for me to be able to pursue my research work.

## Abstract

With the ever-growing dependency on software, testing for their unexpected behavior is as important as verifying for their known properties, to avoid potential losses. Existing software testing approaches assume either the specification as a baseline to test for the intended behaviour of the program's implementation [154] or propose to find ambiguities in the documentation or the specification, with respect to the implementation [115].

In reality, the issue may arise from either of the two sources and hence, it would be more appropriate to build testing techniques to highlight the inconsistencies between the two. There are two limitations that exist in restricting the testing approach to only the implementation and its associated documentation: 1) formulating complete and precise specifications is a hard problem, and 2) test cases giving high coverage on the program may not suffice in ensuring a bug-free implementation. In this direction, the dissertation proposes to leverage resources apart from the documentation and the source code to generate tests to highlight the inconsistencies and understand the introduction of these inconsistencies as a code project evolves.

We leverage existing resources such as test suites from other similar programs, domain knowledge of the developers, external resources such as RFCs etc., for effective test generation. We propose two test generation approaches: 1) Mining existing test suites associated with similar functions in other libraries, to generate test cases. 2) Obtaining a differential model highlighting semantic gaps arising from inconsistencies in an input structure, as inferred from the function's implementation and its associated documentation. These approaches generated tests to reveal defects in real programs, which indicates the effectiveness of leveraging external resources. The first approach was shown to reveal 67 defects through a study on leveraging similar libraries for test generation. These defects were then used to assess the proposed tool to automatically recommend test cases obtained by mining similar functions across open source libraries. The tool revealed 22 defects from the dataset of 67 defects, and additional 24 previously unknown defects, thus, revealing a total of 46 defects. The second approach, based on building a differential model, revealed 80% of the defects in the evaluation dataset and additional 6 previously unknown defects.

We then delve deeper into reasoning about the introduction of such inconsistencies in the process of code evolution where we analyse how the description in a documentation relates to the code changes made at several points in a library. We leverage the commit patterns and the developer discussions to understand the nature of these changes made in the evolution process. We observe relations between methods, such as a call-graph relation, inheritance or interface implementation, and their references in the other method's documentation, to explain the presence of dependencies that can eventually lead to inconsistencies when one entity is modified without updating the dependents.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Chapter 1

# Introduction

In this chapter, we motivate the reader towards the need for software testing. The chapter gives a brief overview of the current practices in the process of testing and the challenges that exist in this process, where we propose to contribute.

## 1.1 The Need for Software Testing

Software testing makes up 30-50% of the software development process [148]. Testing a software requires designing test cases such that passing the test input to the software can indicate if the output complies or deviates from the expected output behaviour. It may cost 10 to 1000 times more, on an average, to fix a bug after the software release than what it would have costed before the release [99]. Hence, it becomes important to design the testing process using an effective and scalable technique.

## 1.2 Testing Process

The process of designing the test cases may be done manually, semi-automatically or fully automatically. While manually designing the test cases may be a tedious task, considering the large space of possible test cases for large programs, it allows leveraging the domain knowledge of the tester in designing quality tests. Semi-automated or automated testing tools may be capable of generating large number of tests, requiring much lesser human intervention than what is needed for manual testing. However, it may be a challenge to detect a defect, especially, if it does not lead to a program crash or violation of formal specifications. Absence of formal specification, which is a common case, may further make the scenario challenging if the test input silently results in a incorrect output, without any program crashes. Hence, the problem of determining the expected output for a given input to the program, which is also referred to as the test oracle problem [16], becomes a bottleneck in automated testing. Such situations call for manual verification over the tests generated by the testing tools. Hence, there is an active interest in techniques to automatically generate a smaller or prioritized set of smart test cases, to reduce the manual load of verification, while minimally compromising with the test coverage [48, 84, 61].

## 1.3 Existing Challenges

Despite the existence of several software testing tools, issue repositories of widely used libraries and applications are flooded with defects reported by users. To investigate where the existing testing techniques lack effectiveness in the practical scenario, we have studied the nature of defects being reported in popular libraries, that are hard to be solved by existing tools. We made two primary observations:

1. A large fraction of defects arise from inconsistency in the documentation and the implementation, often when the specifications are implicit in nature.

2. Developers often assess a test suite on coverage criteria; usually, code coverage is used as the metric.

As inferred from the first observation, a documentation may not be precisely defined. Certain constraints may be implicit, leaving it on to the user's interpretation when using the associated functionality. The specifications derived from the requirements may hence, miss certain details or have ambiguities that may further, be inconsistent with the implementation. For instance, consider the function from Python String Utils library[1] in Listing 1.1 whose documentation states 'Checks if a string is a valid ip'. It appears implicit from the documentation that the validation check is on IP addresses, supporting both ipv4 and ipv6 address types. However, the implementation has support for only ipv4 address.

Furthermore, for the provided example, one can easily obtain a test suite giving 100% coverage by including tests supporting ipv4 address type. However, for a user who relies on the documentation, the function would behave unexpectedly on ipv6 address types, which may be missed by the tester, until reported. The support for ipv6 address validation was added to the library when we reported this issue. Oftentimes, a programmer may end up implementing functionality that differs from specifications on the input, especially, structured inputs such as a file path or URL. These structured inputs, despite being valid as per the

[1]`https://github.com/daveoncode/python-string-utils`

```
def is_ip(string):
    """
    Checks if a string is a valid ip.
    :param string: String to check.
    :type string: str
    :return: True if an ip, false otherwise.
    :rtype: bool
    """
    IP_RE = re.compile(r'^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$')
    return is_full_string(string) and bool(IP_RE.search(string))

def is_full_string(string):
    """
    Check if a string is not empty (it must contains at least one non space character).
    :param string: String to check.
    :type string: str
    :return: True if not empty, false otherwise.
    """
    return is_string(string) and string.strip() != ''
```

**Listing 1.1:** Sample function is_ip to check validity of an IP address

specification, may end up being handled incorrectly in the implementation if logical checks on the expected input are missing or weakly defined or if there are additional checks in the code that constrain the input as compared to its specification in the documentation. Qualitative evaluation of test cases based on code- or path-coverage may overlook such issues.

## 1.4 Problem Definition

Existing software testing approaches have focused on using either the specification as a baseline to test the program's implementation [154] or have proposed approaches to find ambiguities in the documentation or the specification, with respect to the implementation [115]. While the former is a more common scenario, there have been instances of bug reports where the developers chose to fix the documentation for a given program. Furthermore, formulating complete and precise specifications is a hard problem. As a result, there is a need for testing approaches that do not rely on either the specification or the implementation to be correct to use as a baseline to fix the other. Furthermore, there is a need to understand in the first place, how such inconsistencies are introduced, to be able to minimize them.

## 1.5 Thesis Statement

This dissertation confirms the thesis that:

**Thesis Statement**

External resources and meta-data can be leveraged to highlight the gap between a program's implementation and its documentation and understand how these gaps are introduced.

More precisely,

TS-1 Existing resources such as test suites from other similar programs, domain knowledge of the developers, external resources such as RFCs etc. can be leveraged for effective test generation to highlight the inconsistencies between an implementation and its documentation.

TS-2 The meta-data associated with code-projects, such as their commit logs and developer discussion threads can be leveraged to understand the nature and the cause of the introduction of the inconsistencies between an implementation and dependent documentation.

## 1.6 Contributions

The first testing approach, discussed in Chapter 3, proposes a test case mining approach to explore whether there exists similarity among functions across different libraries and languages and whether their associated test suites can be leveraged to reveal defects in one another. Using tests from across libraries allows diversity in the test cases by leveraging the knowledge and expertise of a community of developers, who have spent efforts in writing quality tests and fixing defects in the past. This knowledge may often highlight possible interpretations from the documentation that may be implicit and hence, often overlooked by developers while implementing the function. We assess the potential of the proposed approach for test generation through an empirical study.

Extending from the insights drawn from the study, we propose to automate the idea in the form of a test case recommendation tool, METALLICUS, discussed in Chapter 4. METALLICUS returns a test suite for the given input of a query function and a template for its test suite. METALLICUS incorporates an NLP-based approach, combined with static analysis over test-suites, to extract function matches and maps their parameters to synthesize test-suites for the query function.

The second testing approach, discussed in Chapter 5, aims to expose defects occurring due to input inconsistencies in the function's documentation and the implementation. The idea is to statically analyze the source program to obtain structural constraints on the input to obtain a model. Obtaining another model of constraints from the documentation and external resources such as RFCs, we take a differential between the two models to compute tests that expose the gap between the two. Such an approach would allow highlighting constraints that may not have been defined in the code, that may otherwise be unlikely to be revealed by path-coverage based techniques.

The proposed approaches resulted in the revelation of defects in several libraries, requiring to either fix the implementation or add clarification to the documentation. This indicates the effectiveness of the resources used for a systematic test generation in highlighting the inconsistencies.

Having observed the inconsistencies highlighted through the proposed testing tools, we were motivated to work on reducing the introduction of these inconsistencies in the first place. With this objective, we explored further to understand the causes of the introduction of such inconsistencies in the process of code evolution in a study discussed in Chapter 6. We made use of the meta-data associated with the code-projects where we analysed commit logs to 1) build a taxonomy of the types of documentation updates that indicate dependencies on other changes, 2) obtain patterns that induce indirect dependencies in documentations. These patterns can be applied in building applications to reduce documentation inconsistencies in the context of code evolution by pointing out the dependent places where a change is required given a change made by the developer at another point.

## 1.7 Thesis Publications

### 1.7.1 Chapter 3

**D. Sondhi**, D. Rani, R. Purandare, Similarities Across Libraries: Making a Case for Leveraging Test Suites, IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2019.

### 1.7.2 Chapter 4

**D. Sondhi**, M. Jobanputra, D. Rani, S. Purandare, S. Sharma, R. Purandare, Mining Similar Methods for Test Generation, IEEE Transactions on Software Engineering, 2021.

### 1.7.3 Chapter 5

**D. Sondhi**, R. Purandare, SEGATE: Unveiling Semantic Inconsistencies between Code and Specification of String Inputs, IEEE/ACM International Conference on Automated Software Engineering (ASE), November 2019.

### 1.7.4 Chapter 6

**D. Sondhi**, A. Gupta, S. Purandare, A. Rana, D. Kaushal, R. Purandare, On Indirectly Dependent Documentation in the Context of Code Evolution: A Study, IEEE/ACM International Conference on Software Engineering (ICSE), May 2021.

### 1.7.5 Other publications

(Doctoral Symposium) **D. Sondhi**, Testing for Implicit Inconsistencies in Documentation and Implementation, IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2019.

## 1.8 Outline of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents the background and the related work. Chapter 3 discusses the feasibility of the first test generation approach based on mining test cases. Chapter 4 presents how the proposed test mining approach can be automated and presents its effectiveness in recommending quality test suites. Chapter 5 presents another test generation approach that computes a differential between the input structures inferred from the program's implementation and the documentation. Chapter 6 discusses the insights on the introduction of inconsistencies and the nature of documentation updates that are made as a code evolves, to make a case for an application to recommend updates to minimize such inconsistencies. Finally, conclusion and future work are in Chapter 7.

# Chapter 2

# Background and Related Work

The primary objective of software testing is to highlight weaknesses in a program. These weaknesses may exist in forms such as vulnerabilities, bugs and inefficiencies in terms of runtime, memory (or resource) consumption or scalability. While any form of weakness in a system is seen as undesirable, vulnerabilities and bugs are seen of a serious concern to developers, with respect to security and reliability and hence, proposing techniques to expose these two is of active interest. The dissertation proposes testing techniques to highlight bugs in the context of inconsistencies between an implementation and its documentation. The chapter begins by providing a background on fuzzing (Section 2.1), which is a commonly used automated testing method [79], to highlight the testing techniques proposed over time. The remainder sections present a more focused literature related to the techniques and ideas proposed in this dissertation, in order to relate as well as differentiate between the past and the proposed ideas.

## 2.1 Background

Fuzzing is an approach to automatically generate test inputs to discover bugs or vulnerabilities in a software [22]. A fuzzer (the fuzzing system) handles the end to end testing process. The process comprises of extracting information from the target program and the specification. This information is used to generate the test cases. These test cases are passed as input to the target program to assess its behavior in inferring if it contains a bug. Further, certain prioritization of the detected bugs may be performed by the fuzzing system to filter bugs that are more serious in nature for a certain context. Among all these stages, the test generation stage remains of our primary interest, as it requires the maximum creativity and smartness.

Based on the degree of information that the fuzzer uses from the target program, the fuzzing techniques may be categorised as black box, white box or gray box.

### 2.1.1 Black Box Fuzzing

Black box fuzzing does not take any information from the source code into account, while generating test cases. Random generation of testing cases is the earliest and simplest, yet effective, black box fuzzing technique, as proposed by Miller et al. [92]. A smarter way to proceed is to mutate over a well-formed input based on certain pre-defined rules or prior knowledge to generate malformed test inputs [42, 38].

Black box fuzzers may be capable of producing large number of test cases. However, these techniques involve randomness in the test generation, which may not ensure a high test coverage if a limited number of test cases were to be used for practicality.

### 2.1.2 White Box Fuzzing

Test adequacy is commonly assessed by code coverage criteria such as statement coverage or path coverage [54, 166]. To maximize a certain coverage metric on the code, several fuzzing techniques have been proposed that make use of the source implementation in producing test cases. The information obtained from the implementation could be symbolic constraints obtained from various program paths or data flow information or memory usage or any other detail to refine the test generation. Such testing approaches that rely on analysing the source program are categorised under white box fuzzing. Further, the process of analysing the program may be categorized into static analysis or dynamic analysis. While static analysis infers details from the source code without the need to execute it, dynamic analysis gathers information from the program's behavior observed on its execution.

White box fuzzing techniques aim at covering the target program thoroughly to produce test cases. Symbolic execution is one such technique that aims to gather symbolic constraints from conditional statements in the program along different paths and treats inputs and program variables as symbols to obtain symbolic expressions along respective paths [75, 28, 53]. These expressions are passed to SAT or SMT solvers [14, 73, 8, 65, 80, 37] to generate concrete values that satisfy the expression along the respective path. These concrete values can serve as test values to help developers produce behaviour along a path for the purpose of testing or debugging.

Concolic execution is a dynamic variant of symbolic execution that starts with a concrete input and gathers constraints along the path that the input takes. Iteratively, one of these constraints are then systematically negated to explore other program path constraints to generate more inputs (test cases). Sen et al. propose CUTE and jCUTE [128], concolic execution engines for C programs and Java programs, respectively. DART [52] is another tool that combines random testing with directed search in an attempt to cover all executable program paths.

Symbolic Path Finder [107] uses the analysis engine of Java Path Finder (JPF), which is a model checking tool for Java programs. SPF combines path-coverage based symbolic execution technique with model checking [151]. EvoSuite generates test-suites using a search-based approach which optimizes on coverage criterion [48]. In search-based techniques, search is guided by a fitness function (for example, the execution time of the system under test) to maximize coverage during the test generation process. Seeding refers to any technique that makes use of previous knowledge to help solve the testing problem [123]. This seed helps in the generation process and optimizes the fitness of the population. For instance, a seed could be a test case with high coverage to start the search from. Derakhshanfar et al. propose behavioral modeling learnt from class usages obtained from the system under test and the existing test cases to use as seed [41].

Further, white box fuzzing techniques have been proposed to test programs taking structured inputs. This is particularly useful to test scripts running on browsers that take well-formed input. One such approach derives a symbolic grammar, representing the constraint, from the program and a grammar-based constraint solver returns solution to the constraints [51].

As white box fuzzing allows a higher path coverage as compared to black box fuzzing, it allows diversity in test cases. One shortcoming of path-based approaches is the issue of path explosion that may arise in case of complex programs. In addition, such techniques have a heavy dependency on constraint solvers that may be inefficient or incapable in solving complex expressions.

### 2.1.3 Gray Box Fuzzing

A gray box fuzzing can be seen as a hybrid of black box and white box approaches that gather partial information from the program to design rules to mutate over the test input or the input seed. American Fuzzy Lop (AFL) [160] is a coverage-guided fuzzer that mutates over test inputs using several strategies. If a mutation on input results in discovering a new program state, it is added to the queue to mutate further to generate more test cases. This process is followed iteratively to cover more states. Coverage-based Greybox Fuzzing (CGF) [26] is an extension of AFL. CGF uses a Markov chain model which specifies the probability denoting that fuzzing the input seed generates an input that exercises a different path from that accessed by previous input. Each state (seed) has an energy associated that denotes the number of inputs to be generated from that seed. Based on the observation that most fuzzed inputs access the high-frequency paths of a program, the work focuses to channelize the fuzzing effort towards low-frequency paths to explore more paths with the same amount of test inputs.

Danglot et al. highlight how the recent testing techniques form an emerging field of 'test amplification', a term they coin to describe how the knowledge embedded in existing tests is being exploited to enhance the test cases to attain engineering goals [36]. Some examples of these goals are improvement of coverage of changes or increasing the accuracy of fault localization. Their literature survey identifies four types of approaches to amplification: amplification by Adding New Tests as Variants of Existing Ones; Amplification by Modifying Test Execution; Amplification by Synthesizing New Tests with Respect to Changes; Amplification by Modifying Existing Test Code. Seeding strategies commonly followed by search-based test generation techniques are forms of test amplification techniques [123].

While the software testing community has seen a trend shift from black box to white box to gray box fuzzing [89] in the past few decades, most of the proposed techniques have been evaluated on the metric of path coverage [26, 147, 27, 77, 155]. This metric itself may not be enough to ensure the quality of test cases, especially if certain paths are not well defined in the program, while being present in the specification. Another limitation with most of the existing techniques is that they are specification-based. The complete specifications or program assertions are assumed to be readily available [10, 28, 51, 11], while this may not hold in reality. Derivation of complete and precise specifications is often done manually, making it a tedious task and often prone to errors or incompleteness.

## 2.2 Related Work

### 2.2.1 Testing for Code-Comment Inconsistency

There has been recent focus on defects arising from difference in the code and the associated documentation. Zhou et al. propose a technique to detect semantic defects in documentation using program comprehension and natural language processing [165]. The technique scopes the nature of defects to checks on range limitation and type restriction. It is based on the assumption that the code is correct, which may not be the case, as seen in the listed defects (Table 8).

Tan et al. perform comment analysis to derive program rules, scoped in the domain of locking protocols [141, 143]. These rules are used to detect inconsistencies between comments and source code. @TCOMMENT [144] is another approach for testing code-comment inconsistencies where they propose simple heuristics to analyze free-form text in

Javadoc comments related to method properties for null values and related exceptions in Java libraries.

Alkhalaf et al. present an input validation technique to check input structure inferred from validation function against some given policies [11]. The technique uses dataflow analysis to infer the possible values of the string in the form of a DFA. Alkhalaf et al. run the string analysis over validation functions, hence limiting to exposing input validation defects. SEMREP is a repair tool based on the proposed technique [10]. By analyzing statements in the entire function, one of our techniques (Chapter 5) goes a step ahead to expose deeper functionality flaws, in addition to input validation. While the approach by Alkhalaf et al. checks compliance with certain policies, we derive our own deviation policies through $D_\Delta$. Furthermore, instead of assuming the specification to be correct, we target inconsistencies between the specification and the implementation, which may result in fixing the specification.

Shahbaz et al. propose to explore defects in routines due to missing logical paths [130]. They use web search based approach to gather valid tests and perform mutation over regexes, obtained from web searches, to generate invalid tests. Their approach relies on identifier names to perform web search upon to infer structured values. This may, however, not work for identifiers that heavily depend on the context of occurrence. For instance, identifier name *pair* may refer to credential pairs, JSON key-value pairs or any other pair.

Kim et al. present a grammar-based fuzz testing, called API-level Concolic Testing (ACT) [74]. To improve code coverage of black-box testing, ACT derives a fuzzing grammar from the implementation of the programs taking complex structured input strings. Their approach, however, overlooks the specifications. In Chapter 5, we follow a similar path-coverage based approach to derive the input structure from the implementation, in the form of a regular expression. However, our focus is on generating test strings targeting missing logical paths, for which considering the specifications becomes essential.

Documentation-writing has traditionally been done manually. Recent research has given due attention to automated documentation generation [86, 139]. However, the aspects of their quality and maintenance are equally essential [9, 94, 149, 122, 47, 33]. Aghajani et al. study documentation-related issues reported on different platforms [9]. They build a taxonomy of such issues to suggest actionable points. Our study presented in Chapter 6 complements these findings.

Several refactoring tools have been proposed to minimize code-comment inconsistencies arising from refactoring [118, 108, 115]. Eclipse's Java development tools support automated comment refactoring [108]. While Eclipse's refactoring functionality focuses on lexical matches for renaming identifiers, Ratol et al. propose a lexical and semantic rules-based algorithm to detect fragile comments in the context of performing refactoring-based changes to identifiers in the code [115].

Focusing on the correctness of the documentation, code-comment inconsistencies have been studied in the past and tools have been developed to detect such inconsistencies. Wen et al. study the change-history of GitHub projects to analyze how code and comment co-evolve [153]. They further build a taxonomy of comment-related changes observed on the analyzed commits. iComment is a tool that detects bad comments at a function level [142]. The tool uses a template-based approach by combining NLP, machine learning, and program analysis to highlight gaps in the context of locking protocols. All these studies and techniques are scoped to method-level granularity to analyze the inconsistencies between the method's code and the directly associated comments.

### 2.2.2 Mining-Based Approaches in Software Testing

Several code search approaches have been proposed that mine software repositories [31, 56, 140]. However, retrieval of similar code has primarily been from the perspective of program comprehension, reusability, example retrieval, rapid prototyping, and discovering code theft [81, 15, 124, 88, 87]. While data mining techniques have been proposed for bug detection, the focus has been restricted to code clones [70, 71, 78] or software implemented in the same language. Zhang et al. propose a grafting technique that transplants code stub from one clone to another and uses the resulting mutant to perform differential testing for clones implemented in Java [161]. Carzaniga et al. propose an approach to generate cross-checking oracle by mining intrinsic redundancies in a software [29]. Their technique generates oracles localized around the invocations of methods that admit to redundant alternative code. These oracles then result in potentially fault-revealing cross-checks.

The work by Janjic et al. comes closest to our work [69] on mining test cases, discussed in Chapter 4. However, they propose a proof-of-concept of automating a test recommendation approach, while our work proposes an end to end tool and its evaluation on real dataset of functions. Janjic et al. describe a search engine for unit tests and abstractly propose an Eclipse plugin for test recommendation. Their work relies on method names, class names and the heuristics derived from it to search for candidate tests. They scope the idea to single language as they do not discuss issues pertaining to type compatibility. Furthermore, the work misses discussing the challenges pertaining to parameter mappings required to generate appropriate test recommendations.

Utilizing software mining approaches in the domain of software testing would make a unique class of test generation techniques which has been inadequately explored [40, 12, 145], to the best of our knowledge.

### 2.2.3 Machine Learning for Software Testing

Several techniques have made use of machine learning and natural language processing (NLP) approaches, often combined with program analysis to address software engineering problems such as bug detection [152], behavior verification [55, 142], code completion [116] and specification extraction [163]. Wang et al. propose a bug detection technique based on the assumption that low probability token sequence in a program is indicative of bad practices and potential bugs [152]. They propose an n-gram language modeling on the programs in their bug detection tool called Bugram. The tool learns probability distribution of method call sequences with control flow and flags low probability token sequences as indicative of potential bugs. Gorla et al. make use of NLP, clustering and classification techniques from the machine learning domain to flag API usages by Android apps as normal or abnormal with respect to the app description [55]. We apply NLP and machine learning approaches, discussed in Chapters 3 and Chapter 4, for match extraction and test-suite adaptation.

### 2.2.4 Understanding Code Evolution Aspects

Due to software delivery pressures, developers may be rushed into completing certain tasks, compromising the overall software quality. This phenomenon is referred to as the technical debt [34]. Changes occurring in the process of code evolution involve several risks [125, 35]. Shihab et al. study various factors and their effectiveness in detecting risky changes [133]. One of their findings suggests that developers are unreliable in identifying the risks when changes are made that require related changes. As risky changes incur debt in terms of more reviewing and testing, it becomes essential to identify these risks.

Technical debt may also be intentionally introduced by developers, referred to as self-admitted technical debt (SATD) [111, 68]. This may involve introducing workarounds or sub-optimal code to satisfy the basic requirement, or adding notes for future releases. A study on code-comments suggests that SATD may persist due to the code-comment inconsistent changes after releases [111]. Hence, efforts to warn developers about introducing SATD become essential. The study presented in Chapter 6 is an effort in this direction, by highlighting patterns that can potentially aid in identifying the targets where the changes are needed when a source is updated.

Researchers have studied the development pattern in the context of code-evolution [63] and the impact of code-evolution on the test code [159, 17, 110]. Zaidman et al. study the nature of co-evolution of production code and test code to recognize co-evolution scenarios and share insights relevant to developers and test engineers [159]. Beller et al. present a study to highlight the gap between the expectation and the reality on how testing is done on IDEs, indicating that test driven development is not a popular paradigm [17]. The study makes a case for IDE creators to redesign next generation IDEs to support developers with testing, by integrating features to ease the overall testing process on IDEs. Pinto et al. study how the test suite evolves, especially with the evolution of the application under test [110]. The findings of the study would be useful in building techniques to assist test repairs.

Past studies obtained code evolution patterns to harness for applications such as patch generation [74, 104], language migration [164], code recommendation [101], change guidance [167, 90, 62], and code completion [121]. Nguyen et al. propose a technique to detect semantic code change patterns using their tool, CPATMINER [103]. CPATMINER builds a change-graph from the original and the altered code functions, each represented as a program dependency graph (PDG). The change-graph connects the two PDGs by drawing edges between the unchanged nodes before and after the change. This change-graph is analyzed to mine change-patterns. To infer repetitiveness in change-patterns, CPATMINER checks if the graphs are isomorphic. Mehta et al. propose a correlated change analysis technique that applies machine learning on file commits to mine change-rules based on files that frequently change together [90]. These rules are used to suggest files that should be altered once a change is made in another correlated file. Their tool supports changes at file-level granularity for configuration files. While these techniques focus on learning the changes made to the code and configurations, the study presented in Chapter 6 focuses on code evolution from the perspective of inferring the potential causes and patterns that result in inconsistent documentations due to changes made in other related entities.

# Chapter 3

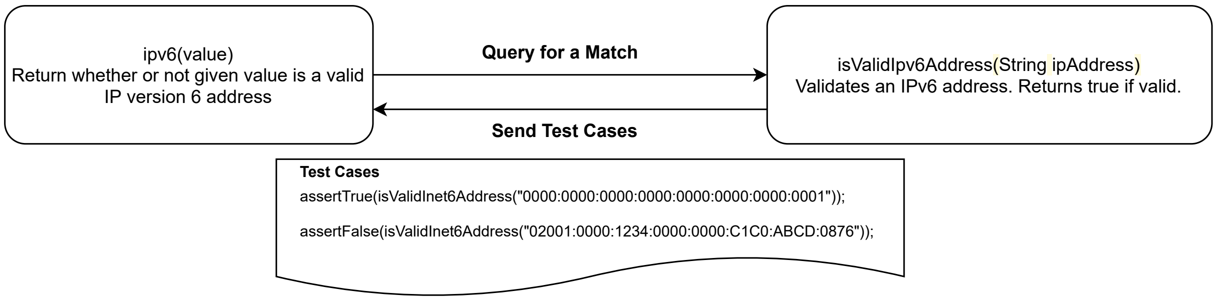
# Leveraging Functional Similarities in Testing: Empirical Study

## 3.1 Introduction

Software applications heavily make use of third-party libraries. Hence, their performance is significantly dependent upon the performance of these libraries, making library testing essential. For an application developer, there may be several options of libraries available that offer functions having similar or exact same functionality. For several reasons, such as computational performance, security, platform or language dependency, accuracy or licensing requirements, one may prefer one library over the other. However, it would be useful to explore whether similarity in functions existing in different libraries can be utilized for testing or for enhancing the functionality and performance of each other.

Developers devote significant time and effort in testing programs using automated or manual testing techniques. However, despite the efforts they are likely to miss certain non-trivial test cases, that may later be raised by users through issue reports. It is a standard practice to maintain test suites for functions supported by libraries. These test suites are updated from time to time by adding test cases on fixing defects reported by users on the issue tracker. Consider two libraries, $L_A$ and $L_B$, implementing the same functionality in functions $F_A$ and $F_B$, respectively. Using test suite for $F_B$ to test $F_A$ may serve as a rich source of test cases. Consider two such functions shown in Fig. 1 that check whether the input string is a valid IP version 6 address and indicate how the test cases associated with one function can be used to recommend test cases for another.

This work appeared at the IEEE International Conference on Software Testing, Verification and Validation (ICST), April 2019


ipv6(value)
Return whether or not given value is a valid IP version 6 address
Query for a Match
Send Test Cases
isValidIpv6Address(String ipAddress)
Validates an IPv6 address. Returns true if valid.
Test Cases

```
assertTrue(isValidInet6Address("0000:0000:0000:0000:0000:0000:0000:0001"));
assertFalse(isValidInet6Address("02001:0000:1234:0000:0000:C1C0:ABCD:0876"));
```

**Figure 1:** Sample of match functions and the test case recommendations

In this study, we have investigated the extent of existence of similar functions across libraries implemented in same or different languages. We have further studied if the tests associated with these functions could be used to reveal defects in other similar functions. Based on the observations, we discuss the challenges in automating this testing process.

While researchers have studied similarities in libraries, in the past, the context has been restricted to library recommendation retrieval, library migration or code migration [162, 146, 102, 72]. To the best of our knowledge, similarities in libraries have not been studied from the perspective of test generation.

In this study, we make the following contributions:

1. **Experiment Design:** To carry out the discussed investigation, we have studied functions in 12 libraries on GitHub[1], spanning over 2 programming languages and 3 library themes. To conduct this study, we have developed a framework that uses a topic modelling based approach on documentations associated with library functions, to extract pairs of similar functions across these libraries. Using the test suites for these functions, we have investigated the potential of the existing test cases in exposing defects in the other library.

2. **Experiment Results:** We observed a significant presence of functionally similar implementations among libraries implemented in same as well as different languages, with a higher percentage of similar functions appearing within a language. On an average, 7.4% of the functions in a queried library resulted in a match in another library.

   Utilizing test suites corresponding to similar functions in different libraries is effective in exposing defects in one another, as evident from the 67 defects that were revealed through this study. 32.8% of these defects were revealed using tests from issue reports submitted in another library. Reported issues and pull requests serve as a rich source in

[1]`https://github.com/`

generating test inputs not only for the library itself, but also for other libraries having similar functions.

## 3.2 Research Methodology

The primary goal of this study is to evaluate the potential of using similarities across libraries by mining similar functions existing in different libraries. By similar functions, we refer to multiple implementations that perform the same or highly similar task. Presence of semantically similar functions in different libraries is indicative of a functional overlap between the libraries. Existence of such similarities could serve as a rich resource for the purpose of code reuse, enhancements, and testing for functionality, security, portability, and performance.

### 3.2.1 Research Questions

We investigated open source libraries available on GitHub, with an objective to get the answers to research questions stated in this section. For several reasons, developers may implement a functionality based on certain requirements instead of reusing the existing functions in other libraries. We studied the extent of prevalence of such a trend by exploring the functional similarities in libraries, within and across languages. This led to the formulation of the first two research questions.

**RQ1:** *How often do similar functions exist across libraries implemented in a language?*

To answer RQ1, we examined the number of functions in a set of libraries for which we could find a similar function, a *match*, in other libraries, implemented in the same language.

Based on our intuition that platform and language requirement may be a key factor in implementing library functions across different languages, we formulated RQ2.

**RQ2:** *Do functionally similar implementations exist across libraries in different languages?*

We studied if, for a given function in a library, there exists a similar function in a library in different language. We further explored if similarity overlaps are more prominent intra- or inter-language and how trends vary across different languages.

We further explored whether the theme of the libraries has an influence on the number of similar functions found between them. Do libraries, complying with a particular theme, result in higher overlap than libraries belonging to other themes? This led to the formulation of RQ3.

**RQ3:** *Does the theme of the libraries influence the number of similar methods found between them?*

Developers maintain test suites to assess the functions in a library. For a given function, having a source of recommendation for test cases would reduce the workload of developers as well as aid in extensively testing a function. This resulted in the formulation of RQ4 to see if similarities across libraries could be leveraged in the domain of testing.

**RQ4:** *Does leveraging test suite for a function result in defect exposure in another similar function?*

Of the similar function pairs, for how many cases using one library's test cases resulted in exposing a defect in the other library's function? What is the nature and popularity of the library that can act as a rich source of test cases? These are some of the questions we investigated in this study.

Well maintained libraries actively maintain reports on issue trackers where developers and users report issues in the library or make pull requests to fix and enhance functions in the library. These reports, often accompanied by test cases to reproduce the issue, may serve as a rich source in improving the library. We investigated if these reports can be useful for other libraries, implementing some similar functions, especially, from the perspective of testing.

**RQ5:** *Can user reported issues and pull requests in a library serve as a source to expose defects across another library?*

To get the answer to RQ5, we investigated the following sub-questions. Of the defects revealed in a function by using other library's test cases for the corresponding function, what fraction of test cases was sourced from a bug report reported by a user of another library? Can bug reports from another library be useful in improving the quality of a library?

### 3.2.2 Data Extraction

To answer the research questions mentioned in Section 3.2.1, we have investigated 12 open source libraries across 2 programming languages and 3 different themes. We chose Java and Python, which are 2 popular, yet diverse languages listed by GitHub as in the year 2017 [4]. Then we picked 3 popular themes that are common across libraries on GitHub for the 2 languages. These themes are- 1) String and numeric manipulation, 2) Natural language processing, and 3) Structural validation. For every theme and a language, we picked two

**Table 1:** Libraries analyzed and the star count of their GitHub repositories (as in September, 2018), indicated in brackets

| | **Theme** | | | | | |
|---|---|---|---|---|---|---|
| **Language** | **String and Numeric Manipulation** | | **Natural Language Processing** | | **Structural Validation** | |
| **Java** | Apache Commons Lang [1,443] | Google Guava [27,127] | Stanford CoreNLP [5,269] | Apache OpenNLP [650] | Apache Commons Validator [79] | Apache Commons IO [504] |
| **Python** | CPython [20,022] | Python String Utils [10] | NLTK [6,914] | Polyglot [967] | Validators [202] | Urllib3 [1,730] |

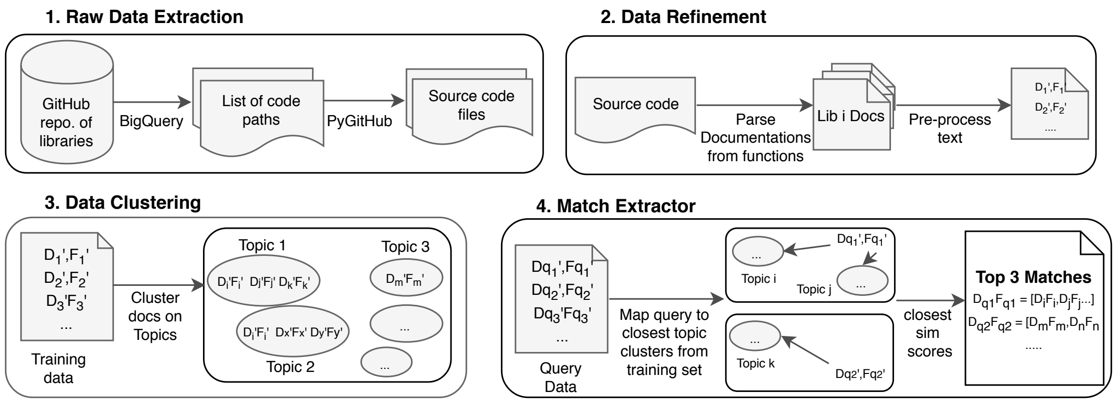


**Figure 2:** Setup of the extractor framework used in the study

projects from a curated list [3, 2]. Following this process, we shortlisted 12 libraries in total, all open source and available on GitHub. Table 1 lists these shortlisted libraries.

#### 3.2.2.1 Extractor Framework

To reduce the manual efforts to search for similar functions in these libraries, we have developed a function extractor framework consisting of the following modules: a) Raw Data Extraction, b) Data Refinement, c) Topic-based Clustering, and d) Match Extractor. Fig. 2 gives an overview of the functioning of different modules in the extractor framework.

**Raw Data Extraction** For a given GitHub repository name as input, the module extracts paths to all source code files from the GitHub repository using the BigQuery API [46]. To avoid inclusion of paths to test code files, we follow a simple heuristic- in the extraction process, we do not include paths that contain the substring 'test'. Using the extracted paths as inputs, the module further extracts the file content using the PyGithub library [5], which is a Python library to access the GitHub API. We used this module to obtain the content of all code files in the 12 libraries.

**Data Refiner** To infer similarity between two functions, we rely on their documentations and the functions' names. By using the documentations that are specified in natural language, we avoid any dependency on the programming language or underlying platform on which a library could be used. Hence, the next task was to extract the documentations and the function names from the file content before proceeding to similarity matching. To extract these components the module uses a) JavaParser [67], for implementations in Java, and b) Python library's ast module for Python implementations.

The module performs the following pre-processing, using NLTK library [19], on every documentation and function name to obtain $D'$ and $F'$, respectively: 1) word tokenization and camel-case splitting 2) stop words and punctuation elimination, and 3) lemmatization.

For future discussions in this section, by *function* we refer to a pair of pre-processed documentation and function name associated with the function.

**Topic-based clustering** The module uses latent Dirichlet allocation (LDA) [21] model to perform topic modelling on the pre-processsed functions in order to cluster functions close to a topic. Topic modelling prioritizes the subset of functions where one is likely to find a similar function, given a query function. The idea of clustering is inspired by Mizarro's framework on relevance which considers documents as relevant to each other if they share a set of common concepts [95, 96]. We denote these 'concepts' by the topics extracted from the documentations. Documentations that are closer to a topic would be more relevant to each other than to those that are closer to other topics. This becomes the basis of clustering relevant, or similar, documentations.

With the task of extracting a potential similarity match for each function, we selected every possible pair of libraries complying with the theme and kept the extracted functions from one library as the test set (or query set) and the other, usually the one with more functions, as the training set. The framework uses Gensim [119], a topic modelling toolkit implemented in Python, to obtain the LDA model from the training set to extract 50 topics, associated with the library chosen for training. Each topic consisted of top 10 high-probability words. Topics-set of size 50 was observed to give the maximum topic coherence score, which is a standard metric used to evaluate the quality of obtained topics. The module maps a cluster of functions in the training set to the closest topic extracted, thus mapping each topic to a set of functions in the training set. Similarly, for every query function in the test set, the module finds the closest topics, lying within a threshold similarity range. As a result, we get a mapping of a cluster of query functions, $C_q$, to a cluster of functions from the training set, $C_t$, for a subset of topics.

**Match Extractor** For each query, we can further explore functions in the corresponding $C_t$, where it is likely to find a match, inferred based on a similarity score. For every query in a $C_q$, the Match Extractor computes cosine similarity on tf-idf vectors, with every function in the corresponding $C_t$. The module computes a pair of tf-idf models from the terms existing in every $D'$ and $F'$, obtained by the Data Refiner, for all the functions in the training set. For a query document and function name pair the tf-idf vector pairs, $tfidf_{doc}$ and $tfidf_{funcName}$, are computed. For every query vector pair of functions in $C_q$, the module computes cosine similarity with every vector pair of functions in $C_t$. The similarity score between two functions is a weighted expression computed as below, with empirically determined weights:

$$sim_{func} = \begin{cases} 0.3 \times sim_{funcName} + 0.7 \times sim_{doc} & \textit{if doc!=null} \\ sim_{funcName} & \textit{if doc=null} \end{cases}$$

For a query function, the functions corresponding to the top 3 similarity scores are picked as a *match-set*, if the scores are greater than an empirically determined threshold of 0.34.

### 3.2.3 Data Analysis

For each of the 3 themes, our dataset contains 4 libraries. We obtained every possible pair of libraries of the same theme, in most cases, assigning the library with more functions for training and the other library for testing. For one library, Python String-Utils, the project description provided on GitHub repository had similarities with the description provided for Apache Commons-Validator and Validators libraries, that were categorized under a different theme. Hence, we also considered the pair of String-Utils with the two mentioned libraries for our analysis, referring to these two pairings as inter-theme pairs.

For every pair, the two library repository names were passed as input to the framework described in Section 6.2.2 to obtain the match-set of top 3 matches for each query function in the test set, also referred to as a query set. We obtained match-sets across 20 pairs of libraries in total. Four volunteers manually validated the correctness of the similarity inference for the returned match-sets. As we considered a set of top 3 functions, for a match, we labelled a match-set as correct (or valid) if at least one function in the set was correct. We discarded incorrect match-sets and analyzed the correct ones. To assess the quality of match extraction, we computed the precision and recall over the extracted matches as defined below.

$$Precision = \frac{\#\ \text{correctly extracted match-sets}}{\#\ \text{of extracted match-sets}} \tag{1}$$

Due to a large unlabelled dataset considered for this study, for recall computation, we assumed that if for a query, a valid match does not appear in top 10 matches, it is not likely to exist in the library used for training. We validated this assumption on 10 randomly picked (query, match-set) pairs from each of the 20 pairs of libraries. We followed a validation procedure that performed a code search on the GitHub repository of the training library by using the key terms (indicative of the topic) extracted from the query documentation. For 95% of these cases, the assumption was correct. Hence, we computed the recall on top 10 matches for every query.

$$Recall = \frac{\#\ \text{match-sets with correct match in top 3 matches}}{\#\ \text{match-sets with correct match in top 10 matches}} \tag{2}$$

For rare cases where a match-set contained more than one valid match functions, we considered the function with a higher similarity score for further analysis. For every correctly matched pair of functions, we extracted the associated test cases for both functions to prepare new test suites by adding assertion checks for the two functions using each others' tests. For two cases, we discarded function pairs from further analyses as neither of them had a corresponding test suite.

The test suite construction required a manual process to convert certain data types or object types to its corresponding type in the other function, taking reference from the template used in the existing test suites. Depending on the complexity of the found match, the efforts and the time to construct the test suite varied. For instance, for scenarios of exact matches of parameters, it is an effort of copy pasting the parameters from one test case to another. For instance, consider the transformation of a test case from one function to another where both functions input a string and escape the Xml characters in the input string.

```
assertEquals("&lt;abc&gt;", StringEscapeUtils.escapeXml("<abc>"));
```

converted to

```
assertEquals("&lt;abc&gt;", xmlEscaper.escape("<abc>"));
```

In this example, the manual efforts mainly depend on the number of test cases existing in the recommended match function's test suite. On the other hand, for complex scenarios where the number of parameters may vary or their type may vary or when the output type may vary, the manual efforts were more involved. In the below instance of the conversion, the tag in the output follows a different format even though both the functions perform PoS tagging. For instance, the tag for proper noun is 'NNP' for one tagger, which is equivalent to 'PROPN' in another tagger. Hence, the output format needed to be appropriately adjusted manually to prepare the assertion.

```
result = pos_tag(word_tokenize("John's big idea isn't all that bad."))
assert result == [('John', 'NNP'), ("'s", 'POS'), ('big', 'JJ'), ('idea', 'NN'),
('is', 'VBZ'), ("n't", 'RB'), ('all', 'PDT'), ('that', 'DT'), ('bad', 'JJ'), ('.', '.')]
```

converted to

```
text = Text( “John’s big idea isn’t all that bad.”)
assert text.pos_tags == [(‘John’, ‘PROPN’), ("‘s", ’X’), (‘big’, ‘ADJ’), (‘idea’,
‘NOUN’), (‘is’, ‘VERB’), ("n‘t", ‘ADV’), (‘all’, ‘X’), (‘that’, ‘DET’), (‘bad’, ‘ADJ’),
(‘.’, ‘PUNCT’)]
```

Consider another complex conversion instance. Consider the function *lastIndexOf(String str)* in Apache Commons-Lang library that takes as parameter a *String str* to search in a *StrBuilder* object (similar to a String). A valid match for this function, *lastIndexIn(CharSequence sequence)* function, in Google’s Guava library, takes a *CharSequence* as input in which to search for a pattern, where the pattern is specified as a *CharMatcher* object taken as an implicit parameter. Hence, the explicit and implicit parameters are swapped in the two cases. In this manner, we used the existing cases in the test suites to infer the template and appropriately prepare the new test suite manually.

We executed the prepared test suites and logged assertion failures. A failed assertion was flagged as a defect only if the documentation associated with the corresponding function did not comply with the observed behavior. To study the source of the defect revealing test case from its original library, we investigated the commit history of the test suite.

## 3.3 Results

Our dataset[2] consists of 31,716 queries, obtained from 12 libraries across 2 languages and 3 themes. Our framework extracted 8,964 matches, having 1,067 valid matches, giving a precision of 11.9% and a recall of 88.6%, using (1), (2).

We observed that for query functions for which a match does not exist in the training library, the extractor still returned some match that contained common key terms, thus, resulting in a low precision but a high recall. Consider a false match returned by our extractor- 1) a query function *withDecimalsParsing* in Apache Commons-Lang library with the query as “with decimals parsing" 2) match returned as function *tryParse* from Google’s Guava library having the documentation as “Parses the specified string as a signed decimal integer value...". ‘decimal’ and ‘parse’ are key terms common in the two strings, causing a match. However, semantically, the query parses a string containing numbers with a decimal point while the returned match function parses a string containing a signed integer with base 10 (referred to as decimal). Consider another pair of query and false match functions, *delete* and *deleteAll.* These functions, though similar in names, are highly generic. For an improved precision, it becomes essential to consider the context, such as class name, where these functions appear, to infer their functionality. We observed a high recall value of 88.6% on the function match extraction. However, we observed scope of improvement in the recall value as well. One key reason why we missed some matches is due to the existence of abbreviations. Consider the terms “greatest common divisor" in one documentation, referred to as “gcd" in another. The current approach may not be accurate in detecting such instances of similarities. The chapter focuses on the study aspect of the similarities in libraries. We discuss the improvement of the extraction technique for tool development in the following chapter.

[2]Data and results available at `https://github.com/pag-iiitd/crosslibTest-study`

**Table 2:** Summary of matches across library pairs. '-' indicates invalid or repeated pair of libraries.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Theme: Natural Language Processing** | | | | | |
| | **Train Library:** | CoreNLP | | NLTK | | OpenNLP | |
| **Test Library** | **# functions** | 17,126 | | 3,668 | | 3,037 | |
| | | **# valid matches** | **# defects** | **# valid matches** | **# defects** | **# valid matches** | **# defects** |
| NLTK | 3,668 | 139 (3.8%) | 18 | - | - | 23 (0.6%) | 4 |
| OpenNLP | 3,037 | 50 (1.6%) | 1 | - | - | - | - |
| Polyglot | 148 | 22 (14.9%) | 2 | 12 (8.1%) | 2 | 16 (10.8%) | 1 |
| | | **Theme: String and Numeric Manipulation** | | | | | |
| | **Train Library:** | Commons-Lang | | CPython | | Guava | |
| **Test Library** | **# functions** | 2,898 | | 13,165 | | 10,333 | |
| | | **# valid matches** | **# defects** | **# valid matches** | **# defects** | **# valid matches** | **# defects** |
| Commons-Lang | 2,898 | - | - | 208 (7.2%) | 4 | 275 (9.5%) | 6 |
| CPython | 13,165 | - | - | - | - | 277 (2.1%) | 8 |
| String-Utils | 24 | 1 (4.2%) | 0 | 4 (16.6%) | 2 | 1 (4.2%) | 1 |
| | | **Theme: Structural Validation** | | | | | |
| | **Train Library:** | Commons-IO | | Commons-Validator | | Urllib3 | |
| **Test Library** | **# functions** | 1,024 | | 657 | | 349 | |
| | | **# valid matches** | **# defects** | **# valid matches** | **# defects** | **# valid matches** | **# defects** |
| Commons-IO | 1,024 | - | - | 7 (0.7%) | 0 | - | - |
| Urllib3 | 349 | 6 (1.7%) | 0 | 9 (2.6%) | 3 | - | - |
| Validators | 32 | 1 (3.1%) | 0 | 7 (21.9%) | 12 | 3 (9.4%) | 2 |
| | | **Inter-theme library pairs** | | | | | |
| | **Train Library:** | Commons-Validator | | Validators | | | |
| **Test Library** | **# functions** | 657 | | 32 | | | |
| | | **# valid matches** | **# defects** | **# valid matches** | **# defects** | | |
| String-Utils | 24 | 2 (8.3%) | 1 | 4 (16.7%) | 5 | | |

### 3.3.1 RQ1: How often do similar functions exist across libraries implemented in a language?

Table 2 lists the number and the percentage of queries, in the test set, for which valid matches were extracted from the training set. For library pairs within a language, we analyzed 7,187 queries in total, with 6,959 in Java libraries and 228 in Python. We found relevant matches for 355 queries, as extracted by our framework. This overlap between implementations in the same language can be favourable for developers from the perspective of code reuse.

**Finding 1:** Similar functions exist in significant numbers across libraries implemented in the same language and theme. 4.9% query functions in our dataset had a match in another library implemented in the same language.

To study whether the extent of overlap depends on the language's constructs and paradigm, we investigated if the share of matches is higher for a particular language in our dataset. Fig. 3 depicts the distribution of valid matches within and across languages for the 3 themes in our dataset. As the number of queries

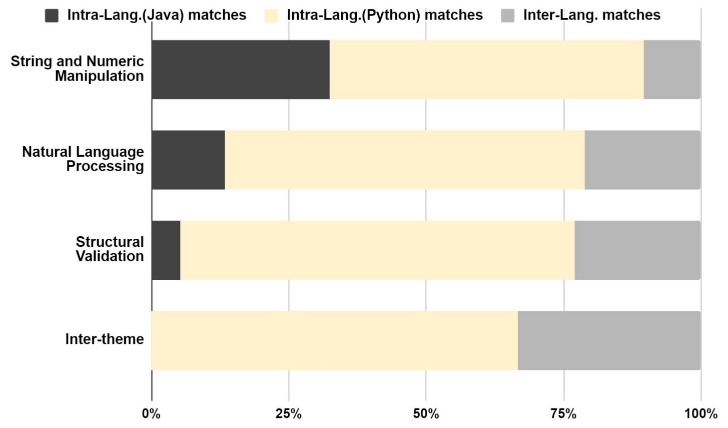


**Figure 3:** Distribution of valid matches for queries within and across languages

across languages vary, the figure presents match values, normalized on the number of queries sourced from a language, per theme. Of the two languages, Java and Python, we observed Python to be consistently dominant, in all themes, in having a higher intra-language library overlap. The trend is indicative of language being an influential factor in the existence of similarities among libraries.

**Finding 2:** Language may influence the extent of existence of functional similarities among intra-language libraries.

### 3.3.2 RQ2: Do functionally similar implementations exist across libraries in different languages?

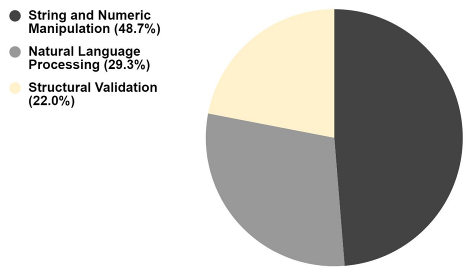
We analyzed the matches obtained for 24,529 query functions across pairs of libraries in different languages. In all, we obtained 712 valid matches, which makes up 2.9% of query to match conversion rate. We observed that even though there exist similarities among inter-language library functions, the extent of this existence is not as significant as in intra-language library pairs. We plan to investigate on this observation further on extending our dataset to more diverse languages, in future.

**Finding 3:** Similar functions exist across libraries implemented in different languages.

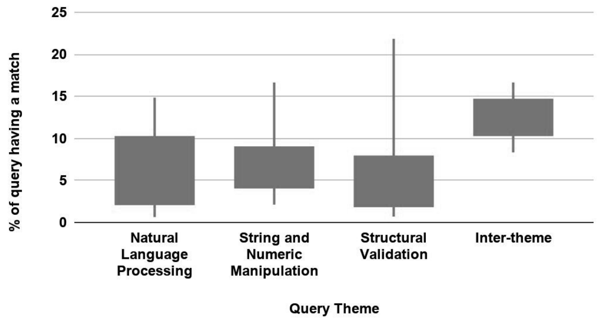
The constructs and the paradigm of a language may be more suitable for certain type of functionalities, causing those functions to exist in one language but rare in the other. One of the key observations explaining the trend of higher overlap within intra-language libraries, relative to inter-language libraries, is the existence of language-specific functions, such as the *parseJavaClassPath* function in Guava library.

### 3.3.3 RQ3: Does the theme of the libraries influence the number of similar methods found between them?

We investigated whether libraries complying with a particular theme result in higher (or lower) number of similar functions than libraries complying with other themes. To analyze a trend on the dependency of



(a) Distribution of valid matches across themes



(b) Distribution of query to match conversion across themes

**Figure 4:** Theme as a factor in finding matches within libraries

similarities on the theme of the library pairs, Fig. 4a shows the distribution of matches across the 3 themes, for all the intra- and inter-language library pairs. The number of matches has been normalized on the number of queries belonging to each theme. While *String and Numeric Manipulation* themed library pairs contributed the maximum share of matches and *Structural Validation* themed library pairs contributed the least, the figure does not indicate a major dominance of any theme in finding the matches. The insignificant effect of the theme in finding overlaps among libraries indicates that the idea of leveraging similarities may be explored for diverse themes. These similarities in functions across libraries may prove to be beneficial for developers of commonly used library functions looking at referring to, reusing or enhancing existing implementations.

**Finding 4:** Theme has an insignificant influence on the extent of similarities among libraries complying with it.

Fig. 4b shows the distribution of the percentage of queries across theme-compliant libraries that were able to convert to a match in another library. We observed a large variation in this conversion rate among libraries complying to a theme. For instance, the conversion rate among libraries themed *Natural Language Processing (NLP)* ranged from 0.6% to 14.9%, with a mean of 6.6%. The queries that did not find a match were usually of functions that were highly specific in nature, often to support another function or specific to the programming language. These specific functionalities may be attributed as the basis for the need to implement the library, instead of reusing the existing libraries. On the other hand, functionalities that are more commonly used, such as PoS-tagging or sentiment analysis in NLP libraries resulted in a match. Our study, nonetheless, resulted in the average query to match conversion rate of 7.4% (standard deviation of 6.2%) across all library pairs in the dataset. This rate is indicative of the potential in leveraging the similarities for purposes such as code reuse, enhancement recommendation, and test generation.

**Nature of similarity in function matches** Below are some of the key observations on the nature of functions that we categorized as similar.

1. *Exactly same functionality*: These functions take the same type of input and give the same type of output following the same implementation approach. Consider the HTML escaping functionality which takes a String as input and escapes the characters in the String using HTML entities. High similarity between functions can be useful for developers looking at code reuse opportunities.

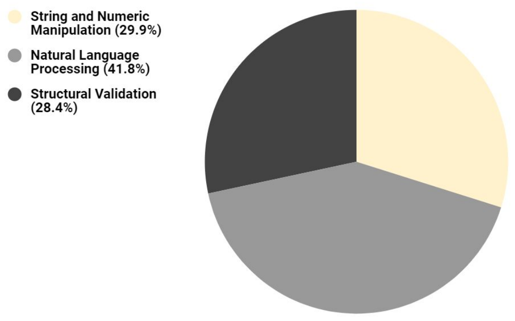
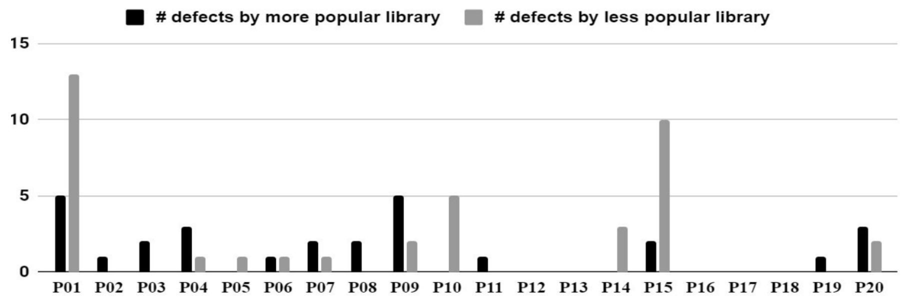
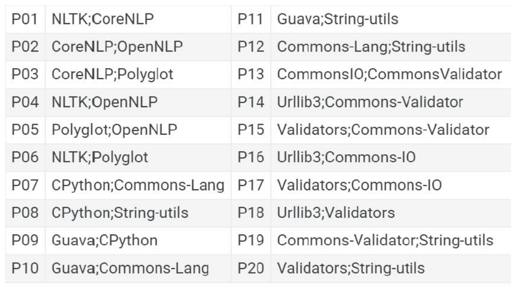


**Figure 5:** Distribution of defects across themes





| | | | |
|---|---|---|---|
| P01 | NLTK;CoreNLP | P11 | Guava;String-utils |
| P02 | CoreNLP;OpenNLP | P12 | Commons-Lang;String-utils |
| P03 | CoreNLP;Polyglot | P13 | CommonsIO;CommonsValidator |
| P04 | NLTK;OpenNLP | P14 | Urllib3;Commons-Validator |
| P05 | Polyglot;OpenNLP | P15 | Validators;Commons-Validator |
| P06 | NLTK;Polyglot | P16 | Urllib3;Commons-IO |
| P07 | CPython;Commons-Lang | P17 | Validators;Commons-IO |
| P08 | CPython;String-utils | P18 | Urllib3;Validators |
| P09 | Guava;CPython | P19 | Commons-Validator;String-utils |
| P10 | Guava;Commons-Lang | P20 | Validators;String-utils |

**Figure 6:** Contribution of defects by each pair of training and testing library. Library name pair convention followed in the plot is <more popular library>;<less popular library>

2. *Superset-subset relation*: We observed certain functions to be a more generic implementation of the matched function. Consider two functions, *join(String separator, int... array)* and *join(Object[] array, final char separator)*, that join elements in an array, separating the elements by a delimiter. While the first function specifically joins integer elements, the second function is a superset that joins all types of valid objects. Such a relation between implementations may be useful from the perspective of improving or enhancing one implementation using another. A subset of test cases associated with the superset function may be useful to test the function in the subset.

3. *High functional overlap but difference in data structures and data types*: Matched functions that use different data types or data structures, such as dictionary in Python and hashmap in Java implementations, to perform the same task, fall under this category. Such variations, despite the functional similarity, could be leveraged by developers to assess the performance aspect of the two functions such as run-time, portability, and security.

4. *Functionally similar but using different algorithm or approach (or APIs)*: Consider NLP themed libraries implementing functions to compute the similarity score between two words. Different libraries contain implementation of several similarity metrics such as path-similarity, Leacock-Chodorow similarity, and Resnik similarity. These variations may be useful for performance assessment of one implementation over the other on metrics such as accuracy and computation cost.

### 3.3.4 RQ4: Does leveraging test suite for a function result in defect exposure in another similar function?

We investigated the usefulness of 1,067 retrieved matches of similar functions in testing. For a match pair of functions, by using test cases available for one function to test the other function, we were able to detect 72 defects in total, resulting from 46 matches. There were instances where one match revealed multiple defects in a function. All these defects have been reported to the developers, of which we have received confirmation of 49 defects so far and 5 defects were rejected as we had misinterpreted the inconsistency due to ambiguity in the documentation. This results in 67 defects from 44 matches.

**Finding 5:** 4.1% of function matches resulted in at least one defect revelation. The observation indicates the potential of leveraging test suites from other libraries for testing.

Table 3 describes few of the defects revealed due to this study. The nature of these defects varied from unhandled cases to weak algorithm followed in one implementation, relative to its functionally similar match, as in the case of the second example on *sentiment analysis* in the table.

Fig. 5 shows the distribution of defects across library themes analyzed in the study. While NLP themed libraries had the highest share of defects (41.8%) revealed by our approach, libraries on structural validation contained the least (28.4%). However, we did not observe a dominance of any theme in defect revelation. We further investigated if a particular set of libraries has a higher contribution in revealing defects over other libraries. Hence, we explored the popularity aspect of the libraries. Based on our intuition that popularity of a library can drive active maintenance of its repository, it may, over the time, lead to improvement of test suites resulting from bug fixes. Hence, we conjectured that the popular libraries may be more beneficial in revealing defects in less popular libraries complying with the same theme.

We quantified popularity of a library by the star count of its GitHub repository, as listed in Table 1. Fig. 6 shows the distribution of defects exposed by each library in a pair, depicted by popularity. Except for 5 pairs of libraries that did not result in any defect revelation, we observed that for 10 pairs, the more popular library of the pair revealed higher or equal number of defects than the relatively lesser popular library. However, for 5 pairs, we observed that lesser popular libraries could also be beneficial in revealing defects in libraries that are more popular.

### 3.3.5 RQ5: Can user reported issues and pull requests in a library serve as a source to expose defects across another library?

To further investigate the sources of effective tests, we analyzed the role of issues and pull requests submitted by users and developers on the issue trackers and GitHub repositories associated with these libraries. We investigated what fraction of test cases, that revealed a defect in another library, were sourced from a reported issue or a pull request. Fig. 7 depicts this fraction value across defects exposed by the 12 libraries that we studied. Of the 67 defect revealing test cases, 22 (32.8%) of them had been added to the test suites either after an issue had been reported by a user or after a pull request to fix a defect had been submitted by a user or a developer.

The reported issues were primarily about corner cases or practically feasible inputs that had been left unhandled by the function implementation. This observation indicates that user-reported issues can be a source of non-trivial tests. Further, as these reported cases were useful in revealing defects in other libraries

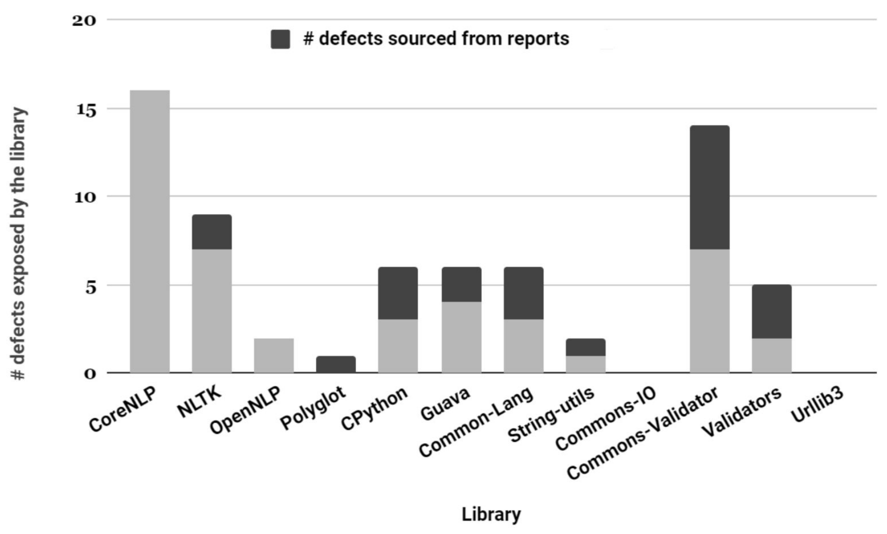


**Figure 7:** Contribution by libraries in revealing defects and the distribution of defects sourced from issues and pull requests

having similarities with the original library, this observation indicates that issue trackers of similar libraries can be a rich source of test cases that developers can leverage. Table 3 includes links to issue reports for applicable cases where the defect revealing test originated from a user report.

**Finding 6:** With 32.8% of the defect revealing cases originating from issues and pull requests submitted by users, issue trackers can be a rich source of non-trivial test cases for functions that are similar to the one in which the defect has been originally reported.

## 3.4 Discussion

The observations indicate that there exist a significant number of similar functions across libraries that can be leveraged to reveal defects in other libraries. Expansion of dataset and using a more precise match extraction approach would help us fetch more matches and further improve defect detection. The results are favourable for developers of new libraries looking at testing their implementations. The results also indicate that existing libraries that undergo active maintenance can make use of these similarities to fix defects and perform enhancements on the existing implementation, by making use of their similar versions available in other libraries.

These observations make a strong case for fully automating the approach followed in this study, which could benefit the developer community. Automation of this software mining based approach to improve the quality of test suites will reduce the load of software testers as well as bring diversity in the test cases, leading to extensive testing of the software. However, automated testing tools based on the suggested approach require consideration of several challenges. Some of these challenges, inferred from the observations drawn from this study, are listed below.

**Table 3:** Some of the defects exposed in functions through the study. For some cases the test case originated from an issue report, link to which has been provided below the defect details, wherever applicable

| No. | Defective function | Test taken from | Defect description |
|---|---|---|---|
| 1. | MTEFileReader. lemma_sents in NLTK | Morphology.lemma in CoreNLP | Lemmatizer cannot handle contractions such as *'ll*, and personal pronouns such as 'her' that should be lemmatized to 'she', but incorrectly returns 'her' |
| 2. | Sentence.sentiment in CoreNLP | SentimentIntensity-Analyzer. polarity_scores in NLTK | The input "Sentiment analysis with VADER has never been this good ." is incorrectly assigned as negative sentiment. |
| 3. | WordUtils.wrap in Commons-lang | TextWrapper.wrap in Cpython | The behavior is inconsistent across the two tests about whether the leading space of string should be retained or not while wrapping. The string " This is a sentence with leading whitespace." retains the leading whitespace over wrap length 50, but removes it at a wrap length 30. |
| | **Link to issue report:** | `https://bugs.python.org/issue15510` | |
| 4. | ipv6 in Validators | InetAddressValidator. isValidInet6Address in Commons-Validator | Validation does not take into account the incorrect presence of extra 0's in the IPv6 address. "02001:0000:1234:0000:0000:C1C0:ABCD:0876" is incorrectly considered as a valid ipv6 address. |
| | **Link to issue report:** | `https://issues.apache.org/jira/browse/VALIDATOR-307` | |
| 5. | is_email in String-utils | EmailValidator. isValid in Commons-Validator | The maximum length of a username or other local-part should be 64 octets in an email. The validator does not have a check on this. |
| | **Link to issue report:** | `https://issues.apache.org/jira/browse/VALIDATOR-362` | |

### 3.4.1 Extraction of similar functions

To be able to use the test suites associated with relevant functions for testing, it is essential to precisely shortlist the relevant pairs of functions. As the approach targets functions in different languages, we used a black-box approach of using documentations for the match extraction, in order to avoid language dependencies. To get the semantically correct pair as a match, it may be beneficial to consider the programmatic features of the implementation as well, which may require the support of program analysis techniques [57]. However, with languages varying in paradigms, introducing language dependency, sourced from a white-box approach, can introduce new challenges.

### 3.4.2 Utilizing test cases and generation of test suites

As observed while constructing test suites, despite the functions being semantically similar, their signatures may have variations. Some of these variations include

- *Difference in input data type:* Consider *CharSequence* input in one function and *String* in another. Essentially, both the input types represent a sequence of characters and the primitive types from the inputs need to be appropriately mapped to generate the test template.
- *Variation in the ordering of parameters:* We observed scenarios where the position of occurrence of corresponding parameters varied across functions $F_1(A_1, B_1)$ and $F_2(B_2, A_2)$. A tool would require applying heuristics to map the appropriate parameters to generate tests.
- *Different number of parameters:* Function matches sharing a superset-subset relation may have variations in the number of parameters they take. Extracting relevant input parameters to generate corresponding tests for another library's function remains a challenge. Further, substituting the input for extra parameter(s) is another challenge.
- *Difference in the source of input:* We observed cases of test inputs sourced from implicit or external sources, other than being specified as explicit function parameters. Consider the match functions *Splitter.on(String separator).split(CharSequence sequence)*, from Guava and *split(String str, String separatorChars)*, from Commons-Lang. While both functions split a sequence of characters on a certain separator, we observe that Commons-Lang's implementation specified the separator input as an explicit parameter, while Guava's implementation takes the separator injected through the *on(String separator)* function. Cases, such as these, make test extraction a non-trivial task in the process of automation.

Further, test frameworks may vary across test suites in different libraries, especially, if the libraries are implemented in different languages. To generate executable test suites, it is required to draw mappings between assertion checks and other statements in test methods, and appropriately inject test inputs in template supported by the corresponding test framework.

## 3.5 Conclusion

We presented an empirical study to analyze the extent of existence of similar functions across libraries. We further explored if test suites associated with similar functions could be leveraged to reveal defects in each other. The study has been conducted on functions in 12 libraries in 2 programming languages and over 3 themes. We observed that an average of 7.4% of the functions in a queried library resulted in a matching function in another library. Using the test suites of functions associated with these match pairs, the study exposed 67 defects. These defects were obtained from 4.1% of the extracted matches. These results are indicative of the potential of leveraging existing test suites for the testing of other libraries containing similar functions. We also analyzed the usefulness of issue reports in finding a use across other libraries. On observing that 32.8% of defect revealing cases, obtained from another library, originated from issue reports, we conclude that issue reports and pull requests can be useful sources in obtaining quality test cases.

These results make a favourable case to build an automated testing tool that could leverage software mining techniques to explore similarities among libraries from the perspective of test generation. Further, extending the scope of repositories to diverse languages and themes would strengthen the effectiveness of the tool.

# Chapter 4

# Testing by Mining

## 4.1 Introduction

In Chapter 3, we discussed that it may be useful to explore whether the similarity in functions existing in different libraries can be utilized for testing or for enhancing the functionality and performance of the functions in either library.

Developers devote a significant amount of time and effort in testing programs, using automated or manual testing techniques. However, despite their efforts, they are likely to miss certain non-trivial test cases, which may later be raised by users through issue reports. It is standard practice to maintain test suites for functions supported by libraries. These test suites are updated over time by adding new test cases, indicative of a bug-fix. These test cases may be sourced from defects reported by users on the issue tracker. Consider two libraries, $L_A$ and $L_B$, implementing the same functionality in functions $F_A$ and $F_B$, respectively. A test suite for $F_A$ to test $F_B$ may serve as a rich source of test cases. Using tests from across libraries allows diversity in test cases by leveraging the knowledge and expertise of a community of developers who have spent effort in writing quality tests and fixing defects in the past. This knowledge can highlight possible interpretations from the documentation, which may be implicit, and hence, often overlooked by developers while implementing the function.

### 4.1.1 Motivating Example

Consider the test method, `testEmailUserName`, on the top in Listing 4.1. The method has been implemented using the *jUnit* test framework in Java and contains a test case to validate the functioning of `validator.isValid`. The function's description states that it checks whether a field has a valid e-mail address. Note that the input string passed to `validator.isValid` is an invalid email address, as the local part in an email cannot exceed 64 characters, while the passed test input comprises of 65 characters in the local part. Hence, the assertion used is `assertFalse`. Consider another test method, `test_invalid_email`, for another function, `email`, shown at the bottom in Listing 4.1. This test method has been implemented in Python using the *Pytest* framework. The description of the `email` function states that it validates an email address, indicating its functional similarity with `validator.isValid`. We adapt the test case from

This work has been accepted in the IEEE Transactions on Software Engineering, 2021

```
public void testEmailUserName() {
    assertFalse(validator.isValid("john56789.john56789.john56789.john56789.john56789.
        john56789.john5@example.com"));
}
```

$\downarrow$

```
@pytest.mark.parametrize(('value',),
    [('john56789.john56789.john56789.john56789.john56789.john56789.john5@example.com',)])
def test_invalid_email(value):
    assert isinstance(email(value), ValidationFailure)
```

**Listing 4.1:** Test suites for functions to check validity of email address

`testEmailUserName` to test case for the `email` function in `test_invalid_email`. If the implementation of the `email` function is inconsistent with the constraint specification on the local part of the email address, it would result in the assertion statement failing on this newly obtained test case. This behaviour would, thus, reveal a defect in `email`'s implementation. We notice that, through such test cases, implicit specifications get highlighted, which help in refining the functionality.

With this proposed idea on mining test cases, one may notice that synthesizing test suites may be challenging. These challenges are sourced from the need for a mechanism to fetch function candidates accurately to retrieve test cases, handling variation in languages and test frameworks, and several other challenges that we discuss in this chapter. While researchers have studied similarities in libraries, in the past, the context has been restricted to library recommendation, library migration, or code migration [162, 146, 102, 72]. To the best of our knowledge, similarities in libraries have not been studied from the perspective of test mining.

**Our Contributions:** Based on the observations discussed in Chapter 3, our extended contribution is the identification of challenges involved in automating this testing process and the proposal of an automated test recommendation tool, Metallicus. Metallicus returns a test suite, populated with mined test-cases, for the given input of a query function and a template for its test suite. The query consists of signature of the function, documentation describing the functionality, and its entities, such as parameters and the return value. We make the following substantial additions to our previous work.

1. An improved match framework, with the incorporation of a match classifier and a two-step refining process. The second stage of refining leverages the nature of concrete parameter values in test samples, to shortlist a match function.

2. A fully automated test recommendation tool, Metallicus, based on the prototype assessed in the empirical study. Each module in Metallicus (shown in Fig. 8) is an added contribution and was not proposed in the previous work, where the steps currently performed automatically by Metallicus were either done semi-automatically or manually.

3. Re-conducting experiments to assess Metallicus on the prepared evaluation dataset and revelation of new defects through a newly added experiment.

4. Identification of challenges that need to be overcome in the future to improve the effectiveness and performance of Metallicus. These insights make a case for open research problems in the area of test mining.

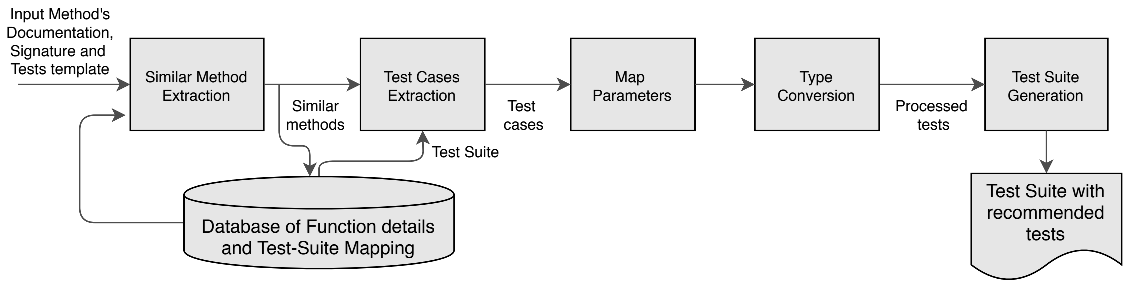
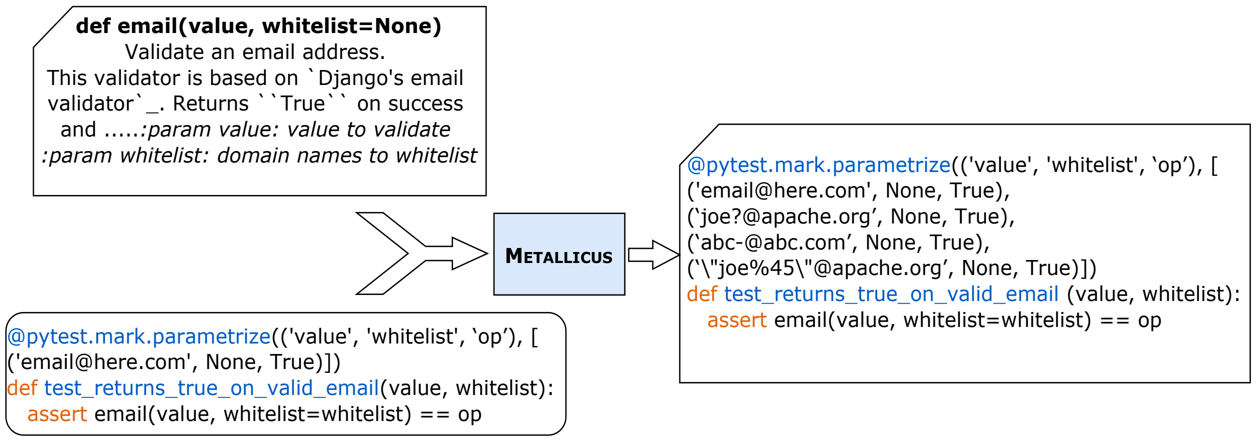

Input Method's Documentation, Signature and Tests template
Similar Method Extraction
Similar methods
Test Cases Extraction
Test Suite
Test cases
Map Parameters
Type Conversion
Processed tests
Test Suite Generation
Test Suite with recommended tests
Database of Function details and Test-Suite Mapping

**Figure 8:** Metallicus's Architecture



```
def email(value, whitelist=None)
Validate an email address.
This validator is based on `Django's email
validator`_. Returns ``True`` on success
and .....:param value: value to validate
:param whitelist: domain names to whitelist
```

```
@pytest.mark.parametrize(('value', 'whitelist', 'op'), [
('email@here.com', None, True)])
def test_returns_true_on_valid_email(value, whitelist):
  assert email(value, whitelist=whitelist) == op
```

```
@pytest.mark.parametrize(('value', 'whitelist', 'op'), [
('email@here.com', None, True),
('joe?@apache.org', None, True),
('abc-@abc.com', None, True),
('\"joe%45\"@apache.org', None, True)])
def test_returns_true_on_valid_email (value, whitelist):
  assert email(value, whitelist=whitelist) == op
```

**Figure 9:** Working Prototype of Metallicus

Metallicus does not depend on the source code for analysis. As a consequence, the tool may be useful for developers who are interested in implementing a functionality, but they would like to gauge the requirements of the function, through the test cases, before implementing. Metallicus may furthermore be useful for developers who have already written a program and are interested in obtaining inputs to test it.

## 4.2 Tool Architecture

The observations made in the study discussed in the previous chapter could be beneficial for developers of new libraries who would like to test their implementations. The results also indicate that libraries that undergo active maintenance can fix defects and enhance their existing implementation by leveraging their similar versions available in other libraries. These observations make a strong case for automating the approach presented in the study, which could benefit the developer community.

Addressing the challenges identified in Chapter 3, we discuss the architecture of the automated tool, Metallicus. As illustrated in Fig. 9, Metallicus expects two inputs: 1) the query function's description along with its signature, and 2) a template of the test suite to indicate the format in which Metallicus should adapt to a test suite. As the output, Metallicus returns a test suite populated with recommended test cases for the given query (right side of Fig. 9). Fig. 8 depicts the internal components of Metallicus. The tool is divided into different modules, each of which has been described in this section.

**Table 4:** Classification report of the match classifier

| | **Precision** | **Recall** | **F1-score** | **Support** |
|---|---|---|---|---|
| 0 | 0.82 | 0.89 | 0.85 | 708 |
| 1 | 0.89 | 0.82 | 0.85 | 786 |
| accuracy | | | 0.85 | 1494 |
| macro avg | 0.85 | 0.85 | 0.85 | 1494 |
| weighted avg | 0.86 | 0.85 | 0.85 | 1494 |

### 4.2.1 Function Match Extraction

The objective of this module is to return the function matches from the database of libraries for a given query function. We used the database of 12 libraries analyzed in the study. As the match framework used in the empirical study (Chapter 3) had scope for improvement, we propose another version of the match framework in Metallicus. To shortlist a set of potential candidates, we performed topic-based clustering on the database of 19,939 source functions. For an input query, Metallicus shortlists the relevant clusters and further evaluates all the functions in those clusters.

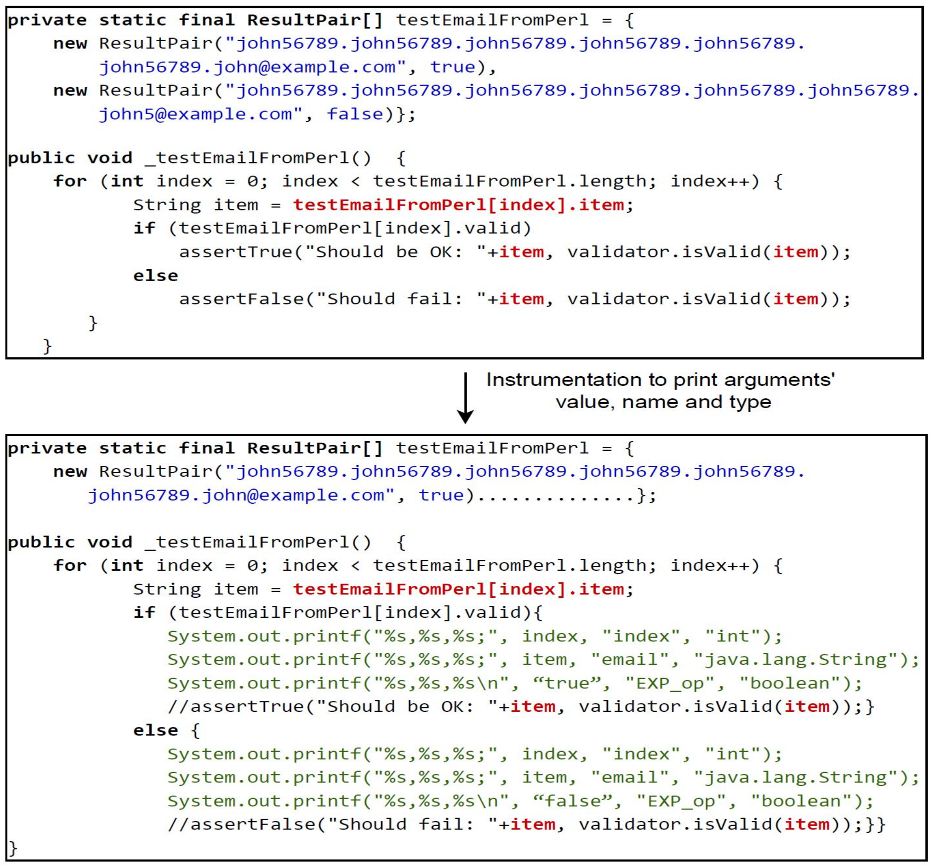
To extract relevant function matches for a given query, Metallicus relies on two sources of information to compute a match score between the query and the candidate function: 1) the associated documentation that is available in natural language, and 2) meta-data associated with the function, which includes the function's signature, which is comprised of the function's name and the parameters passed to it. The documentation is further split into two features; the function description is extracted from the documentation and the description of the parameters is treated as another feature. In all, Metallicus considers three features to compute a match classification: 1) the function's description, 2) the parameters' description, and 3) the function's name. These features are available in natural language, irrespective of the programming language of the implementation source. Metallicus applies natural language processing techniques to obtain a set of three similarity scores between the query's features and the candidate function's features: 1) $sim_{descr}$, 2) $sim_{param}$ 3) $sim_{name}$.

***Word embeddings to compute text similarity*** As a one-time pre-processing step, we obtained word embeddings using Word2Vec (CBOW) model [91]. The embeddings have been trained on pre-processed tokens, with a vocabulary of 19.4k words. The vocabulary has been obtained from set of documentations from libraries used in the study (Chapter 3), dump from *computersciencewiki.org*, javadocs from IJDataset[1] and snippets from RFCs. We picked domain-specific corpus to obtain the word embeddings because Metallicus processes text that is specific to the programming domain. The domain may contain words such as `int` for an integer type or `CharSequence`.

Every feature text undergoes pre-processing of word tokenization, stop-word elimination, camel-case splitting, punctuation removal and lemmatization.

For every feature pair between the query and candidate, Metallicus uses the *Gensim* library [119] to compute a soft cosine similarity score [134] between the text pairs. Each text is modelled into a matrix from the word embeddings. The intuition behind soft cosine similarity is to use the semantic similarity between the words, which can be derived from Word2Vec, to compute a similarity measure between two text pieces. The

[1]`https://github.com/clonebench/BigCloneBench`



```
private static final ResultPair[] testEmailFromPerl = {
    new ResultPair("john56789.john56789.john56789.john56789.john56789.
        john56789.john@example.com", true),
    new ResultPair("john56789.john56789.john56789.john56789.john56789.john56789.
        john5@example.com", false)};

public void _testEmailFromPerl()  {
    for (int index = 0; index < testEmailFromPerl.length; index++) {
           String item = testEmailFromPerl[index].item;
           if (testEmailFromPerl[index].valid)
               assertTrue("Should be OK: "+item, validator.isValid(item));
           else
               assertFalse("Should fail: "+item, validator.isValid(item));
       }
   }
```

```
private static final ResultPair[] testEmailFromPerl = {
    new ResultPair("john56789.john56789.john56789.john56789.john56789.
       john56789.john@example.com", true).............};

public void _testEmailFromPerl()  {
    for (int index = 0; index < testEmailFromPerl.length; index++) {
           String item = testEmailFromPerl[index].item;
           if (testEmailFromPerl[index].valid){
              System.out.printf("%s,%s,%s;", index, "index", "int");
              System.out.printf("%s,%s,%s;", item, "email", "java.lang.String");
              System.out.printf("%s,%s,%s\n", “true”, "EXP_op", "boolean");
              //assertTrue("Should be OK: "+item, validator.isValid(item));}
           else {
              System.out.printf("%s,%s,%s;", index, "index", "int");
              System.out.printf("%s,%s,%s;", item, "email", "java.lang.String");
              System.out.printf("%s,%s,%s\n", “false”, "EXP_op", "boolean");
              //assertFalse("Should fail: "+item, validator.isValid(item));}}
}
```

**Figure 10:** Static analysis and instrumentation for test case extraction

computed scores, $sim_{descr}$, $sim_{param}$ and $sim_{name}$ are passed to a pre-trained Decision-Tree classifier that classifies whether the candidate is a match to the query. A labelled dataset was prepared with the three types of similarity scores obtained on 1489 function matches from the dataset of functions used in the study. The classifier was trained on this dataset. The distribution of the dataset was of 708 positive (valid) matches and 781 negative (invalid) matches. Table 4 shows the classification report on the trained set, where 0 indicates a positive (valid) match and 1 indicates a negative (invalid) match. F1-score of 85% was obtained on the valid matches in the trained set. We performed a stratified 5-folds cross validation on the classifier model to evaluate the quality of training. The cross validation resulted in mean precision of 0.82, mean recall of 0.81 and mean F1 score of 0.81.

For all the candidates classified as matches, Metallicus fetches the corresponding test suites from the database and passes to the test extraction module. A matched function which does not have a test suite written for it is discarded from further processing.

### 4.2.2 Test Case Extraction

Almost all popular open-source libraries contain a test suite associated with a function that lists test cases on which the function has been tested. The essential components of a unit test case are the set of inputs to pass to the function under test and the expected output. The objective of this module is to extract the matched function's input parameters and the expected output from the test assertions present in the test suite of the matched function.

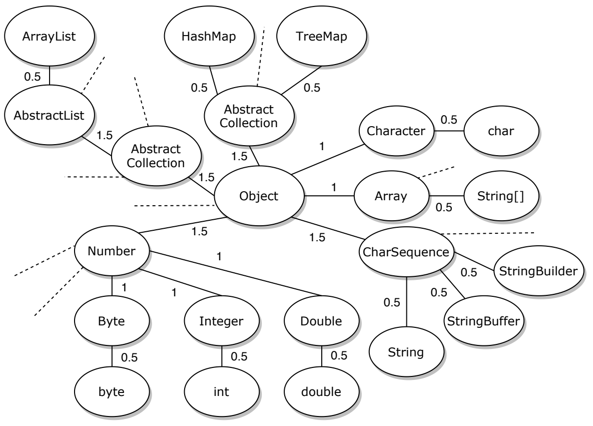


**Figure 11:** Snippet of Java‘s data type hierarchy

One challenge identified in the study (Chapter 3) is to identify implicit and indirect function parameters, if there are any. These parameters may appear from indirect sources, commonly through constructor arguments. The challenge is when these implicit parameters appear as explicit parameters in the query function, and hence, their test values are important to be captured.

The module performs data-flow analysis to statically analyse the test methods of the matched function. The analysis captures all possible variable references or concrete values influencing the test assertion statements. Note that the list of these influential values would essentially contain the explicit and implicit function parameters and the expected output. Consider Fig. 10, where the snippet on the top shows a test suite implemented using the *jUnit* framework to test a matched function, `isValid`. Consider how the assertion statements do not have the test inputs directly passed as parameters to the function. The parameter is a reference to a String object, `item`, which further refers to the element of an array, `testEmailFromPerl`, of `ResultPair` objects. With such a flow of references, it is non-trivial to statically determine the concrete values of the test case. Hence, the module applies static analysis as described, followed by static instrumentation to capture the concrete values. The module captures all references, indicated in red (Fig. 10), which influence the assertion statement.

The module instruments the test suite to print the values of variables captured by the static analysis, along with their variable names (if present) and their data type, as shown at the bottom in Fig. 10. The variable names and data types are captured to be used by other modules of Metallicus at a later stage (details in Section 4.2.3). The instrumented test suite is executed to log the tuples of *(value, variable name, data type)* for every test case. Note that while capturing the data type of a variable in an implementation in Java is possible at static time, Python is a dynamically-typed language. The execution of the instrumented code has an added purpose apart from extracting the concrete values; it allows the capture of data types of variables in an implementation in Python. Additionally, the assertions, normally found to be invoking the target function under test, are commented out while instrumenting in order to avoid execution of intensive functions calls that are not required at this stage.

Metallicus currently supports two languages, Java and Python. It uses the *Soot Framework* [76] to analyse test suites implemented using Java’s *jUnit* test framework, and the *ast* module in Python to analyse test suites implemented using the *unittest* and *Pytest* framework. Since the analysis is generic and light-weight, the support may be extended to other languages and frameworks.

### 4.2.3 Parameter Mapping and Type Conversion

After extracting the test cases, the next stage is to address the mismatch in the number of parameters and the difference in the ordering of parameters with respect to the query function. For this purpose, the parameters from query function need to be mapped to the closest parameter (explicit or implicit) of the matched function. These parameter matches are decided based on: 1) similarity in the parameters' description obtained from the documentation, 2) similarity in the parameters' names, 3) similarity in the nature of their values, and 4) similarity in their data types.

***Finding similarity in the nature of parameters*** The *nature* of a parameter value is inferred in terms of its length, type of characters (numeric, alpha-numeric or special), and the abstract structure. For instance, abstract structure for an instance of ipv4 address looks like *N3.N3.N3.N3*, where *N* represents a numeral followed by its frequency and dot ('.') is a special character, and is hence not abstracted. The intuition is that parameters with similar structures would likely map to each other. However, in order to determine the nature of the parameters of the query and matched functions, METALLICUS requires their concrete values. These concrete values for the matched function are obtained from the extracted test cases. The concrete values for the query function are obtained from the test template that the user provides, which is assumed to contain at least one sample of the test case.

***Finding similarity in the data types of parameters*** METALLICUS makes use of a graphical representation of type hierarchy supported in a language to compute a similarity score between two data types. Fig. 11 shows the snippet of the graph for Java's core object types and primitive types. It is an undirected weighted graph. The edges having one end as a non-internal node are assigned a weight of 0.5; edges having one end as an *abstract class* or *interface* type of node are assigned a weight of 1.5; the rest of the edges are assigned a weight of 1. The intuition behind assigning uneven weights is that as one moves up in the type hierarchy, the data types lose the concreteness and the semantic properties at a higher rate. For instance, `String` should be closer to `StringBuilder` than to `int` type, when computing the distances. The similarity score between two data types is computed as an inverse of shortest weighted distance, applying Dijkstra's algorithm, shown in the formula below:

$$sim_{type} = \frac{1}{1 + \gamma \times dist}, \text{ where } dist = shortest\_weighted\_distance(graph,\ datatype_1,\ datatype_2)$$

$\gamma$ is set to an empirically determined value of 0.4. A graph, similar to the one shown in Fig. 11 is maintained for Python, as can be done for other languages as well. Two of the authors performed the task of linking semantically similar nodes (denoting data types) across languages to form one common graph. For instance, a java_boolean is equivalent to python_bool or java_HashMap's closest equivalent in Python is python_dictionary. Graph-building is a one-time exercise, once support for a language is added to METALLICUS. Code clones-based approaches for token mapping across languages have been proposed recently [24] which obtain shared embeddings on code tokens. Such techniques may be useful in automatically drawing mappings between data types across different languages, provided a cross-language dataset of clones is available. We would be interested in leveraging such an approach as we add support for more languages in the future.

Similar to the classifier used in Section 4.2.1, we trained a Decision-Tree classifier that classifies the closest mappings between parameters according to the expression:

$$Map(par_{i\_query}, par_{j\_match}) = Classify(sim_{arg\_name}, sim_{type}, sim_{par\_descr}, sim_{test\_feat})$$

Parameter pairs that are classified as positive mapping and have maximum classification probability are shortlisted.

Based on the obtained parameter mappings, the test input parameters are arranged in the order of appearance of mapped parameters in the query function's signature. When a parameter from the query function does not obtain a mapping from the matched function, METALLICUS picks the default value obtained from the test template provided by the user to use as the parameter input.

Another challenge is to convert the parameter (or expected output) to an appropriate data type, especially when the mined test suite and the test suite to be adapted to are implemented in different languages. For instance, a `null` in Java corresponds to `None` in Python or a string could be represented within single quotes in Python, while Java represents a string in double quotes. METALLICUS maintains several such conversion mappings for primitive and common types to appropriately convert the values.

***Parameter features for refining function matches*** This module also performs the second stage of match refinement of the function candidates that are shortlisted for the query function. The intuition behind refinement is that if none of the parameters of the matched function has a closely similar 'nature' with any of parameters of the query function, then it is highly likely a false function match. Hence, such a match is discarded from further processing.

### 4.2.4 Test Suite Adaptation

The objective of this module is to populate the template of test suite provided by the user with test assertions containing the mined and processed test cases. METALLICUS currently supports adaptation to test suites in Java's *jUnit* framework and the *Pytest* framework for Python. To generate a *jUnit* test suite, METALLICUS uses *JavaParser* to extract assertion templates and inserts similar assertion nodes to the test suite with new test cases injected. For test adaptation to *Pytest*, a similar test case injection is done using the *ast* module in Python. METALLICUS returns the populated test suite as the final output to the user.

## 4.3 Evaluation

We assessed METALLICUS on 1) its effectiveness of match retrieval, 2) the quality of test cases recommended based on the defects revealed by it in the query function, and 3) code coverage on the query function's source code by the recommended test cases[2].

All evaluations were performed on a Windows 10 64-bit system with an Intel Core i7-5500U 2.40GHz processor with 8GB RAM. The tool has been built and executed with Java 1.8 and Python 3.7.

### 4.3.1 Experiment Setup: Dataset A

We prepared an evaluation dataset of query-match function pairs. We shortlisted the query functions in which defects were revealed through the empirical study (Chapter 3). Of the 67 defects sourced from 44 pairs of query and matched functions, we shortlisted queries such that 1) the defect revealing test case is sourced from a test suite of a matched function (other sources of test cases may be issue reports or examples in the

[2]Tool, data and results available at `https://github.com/pag-iiitd/Metallicus`

**Table 5:** Performance summary of METALLICUS on the evaluation dataset A

| **Performance Metric** | **Value** | | |
|---|---|---|---|
| Execution time (in sec.) per query | Average: 273.3 | Stdev.: 262.0 | Range: 39.8 - 890.4 |
| No. of match-pairs detected from the evaluation set | 15 (of 15) | | |
| No. of new match-pairs detected | 13 | | |
| Match Precision | 87.5% | | |
| Match Recall | 90.3% | | |
| No. of valid matches returned per query | Average: 2.3 | Stdev.: 1.1 | Range: 1 - 5 |
| No. of cross-language valid matches detected | 17 | | |
| No. of test cases generated per query | Average: 204.3 | Stdev.: 191.7 | Range: 7 - 502 |
| No. of total defects revealed | 41 | | |
| No. of known defects revealed | 22 (of 67) | | |
| No. of new defects revealed | 19 | | |
| Statement coverage obtained by tests on query | Average: 95% | Stdev.:8.6% | Range: 75 - 100% |

function documentations etc.), and 2) the test suite of the matched function uses one of the test frameworks supported by METALLICUS.

We shortlisted 22 defects sourced from 12 unique queries. Of the 12 queries, 4 queries are sourced from libraries written in Java and 8 queries are from libraries implemented in Python. Hence, their test suite templates are implemented in test frameworks supported by their respective languages. Furthermore, the 22 defects are sourced from 15 unique query-match function pairs.

For each of the shortlisted query functions, we fed two details to METALLICUS: 1) its associated documentation, and 2) a template of the test suite containing one sample test case (assertion) statement.

### 4.3.2 Results: Dataset A

Table 5 gives the summary of METALLICUS's performance when assessed on the evaluation dataset. METALLICUS executed for a reasonable average duration of 277 seconds per query to return the test suite containing test recommendations. For every query, the execution time has been averaged over 10 runs. Note that we performed test extraction from all functions in the repository beforehand, in order to store the test cases and their associated metadata in a language-independent format in a text file. We did this to avoid the computation overhead in performing static analysis and instrumentation on the candidate's test suite for test extraction. This is particularly useful for avoiding redundant computations for cases when a match appears frequently in the shortlists. As this is a one-time extraction effort, we have not included its time in the execution time of METALLICUS for a query input.

#### 4.3.2.1 RQ 1: *How effective is* METALLICUS *in shortlisting relevant function candidates for test mining?*

Of the 44 defect-revealing match pairs obtained in the study (presented in Chapter 3), we shortlisted 15 pairs based on the criteria described in Section 4.3.1. As indicated in Table 5, METALLICUS could successfully detect all 15 matches while additionally reporting 4 false matches in total. Having a large size of unlabelled dataset, we used the same formula to compute the recall of the match extractor component of METALLICUS, as used in the study. Eq. (2) in Chapter 3 states this formula to compute the recall on top 10 matches.

Metallicus returned 13 more valid matches for the given query set, resulting in a precision of 87.5% and a recall value of 90.3%. This makes the total number of match pairs in our evaluation set as 28, of which 17 pairs were cross-language. *Cross-language* match pair implies that the language of the test template is different from the language of matched function's test suite.

For a given query, there may be multiple match candidates possible, as was observed for over 75% of queries. On an average, the database of libraries linked to Metallicus returned 2 valid matched functions for a query. A high number of matched functions allows higher diversity in the test cases recommended from their associated test suites.

#### 4.3.2.2 RQ 2: *Are the tests recommended by* Metallicus *effective in revealing defects?*

For every query, we executed the test suite returned by Metallicus by integrating the implementation of the function corresponding to the query. For every assertion failure, we manually validated whether it indicated an inconsistency with the documented or expected behaviour. Metallicus resulted in the recommendation of test cases revealing 41 defects (Table 5) for the given set of queries. We further discuss the distribution of these defects from among the previously known cases and the newly discovered defects in the following sub-questions.

**RQ 2a:** ***How effective is*** **Metallicus** ***in revealing the defects exposed through the empirical study?*** Of the 67 defects revealed in the empirical study, we shortlisted queries that could potentially reveal 22 defects. Metallicus adapted test cases to reveal all 22 defects.

**RQ 2b:** ***Does*** **Metallicus** ***result in revealing new defects that the manual process followed in the empirical study failed to reveal?*** With complete automation of the technique, test suites could be adapted without the manual intervention for type conversion and test injections. Furthermore, with an improved match extraction functionality of Metallicus, more match candidates were revealed for the queries used for evaluation. This resulted in test cases that revealed 19 new defects in the query functions which had been missed while conducting the empirical study. These defects have been reported to the developers, of which 8 defects have been validated so far.

### 4.3.3 Experiment Setup: Dataset B

We also evaluated Metallicus on a dataset of query functions obtained from three external libraries, two written in Java and one in Python [3] [4] [5], which implies that they were not part of the empirical study. Hence, there was no set of previously known match candidates or defects for these query functions. These are three open source libraries, one each for the supported three themes of string and numeric manipulation, natural language processing, and structural validation. For each of the 17 query functions present in these libraries, we input Metallicus with their associated documentation and a template of the test suite.

Note that one of the use-cases of Metallicus is for a developer who would like to gauge the function's requirements through test cases before implementing it. In such a scenario, the documentation would not be

[3]`https://github.com/giorgosart/java-string-utils`

[4]`https://github.com/clips/pattern`

[5]`https://github.com/tomansill/JavaValidation`

**Table 6:** Performance summary of Metallicus on the evaluation dataset B

| Performance Metric | Value | | |
|---|---|---|---|
| Execution time (in sec.) per query | Average: 226.8 | Stdev.: 257.4 | Range: 28 - 1039 |
| No. of new valid match-pairs detected | 26 | | |
| Match Precision | 68.4% | | |
| No. of valid matches returned per query | Average: 1.5 | Stdev.: 1.3 | Range: 0 - 5 |
| No. of test cases generated per query | Average: 7.5 | Stdev.: 7.8 | Range: 3 - 27 |
| No. of new defects revealed | 5 | | |
| Statement coverage obtained by tests on query | Average: 93.8% | Stdev.:14.4% | Range: 50 - 100% |

readily available. To capture the manual efforts required in preparing the input to Metallicus, especially in the absence of the documentation, we conducted a user study among 6 external participants, each having at least three years of development experience. The study was conducted on 9 functions to be used as queries, with 3 functions assigned to each participant and each function being assigned to 2 participants. For every function, the participant was provided with its source and a blank unit test template for reference. Each participant was asked to prepare documentation and a test case. They were given time to understand the functionality and discuss with other participants to confirm their understanding; this was done to simulate the understanding of the developers. We then captured the time taken by each participant to prepare the input. We observed input preparation average time of 5.8 minutes, with standard deviation of 5.9 minutes and median of 4 minutes.

### 4.3.4 Results: Dataset B

Table 6 summarizes the performance of Metallicus on the 17 queries passed from the external libraries. Metallicus executed for a reasonable average duration of 227 seconds per query to return the test suite containing test recommendations. For every query, the execution time has been averaged over 10 runs. As followed while evaluating on dataset A (Section 4.3.2), the reported time does not include the time spent in the static analysis and instrumentation as that stage is performed by Metallicus beforehand, as a one-time task.

#### 4.3.4.1 RQ 1: *How effective is* Metallicus *in shortlisting relevant function candidates for test mining?*

Metallicus returned 26 valid match functions for the 17 query functions. A single query may find multiple match candidates; we observed an average of 1 match returned per query, however, the overall number of valid matches ranged from 1 to 5 for a query function taken from dataset B. For 3 of the 17 queries, Metallicus did not return any match. To investigate about the presence of a match for these 3 queries, we passed a SQL query to the database of functions linked to Metallicus to return function documentation that contain keywords from the query function's documentation. The SQL query did not result in any relevant match present in the database, indicating absence of a match. This observation indicates that Metallicus can minimise false positives effectively.

#### 4.3.4.2 RQ 2: *Are the tests recommended by* Metallicus *effective in revealing defects?*

For every query, we executed the test suite returned by Metallicus by over the function corresponding to the query. For every assertion failure, we manually validated whether it indicated an inconsistency with the documented or expected behaviour. Metallicus resulted in the recommendation of test cases revealing 5 previously unknown defects (Table 6) for the given set of queries. These defects have been reported to the developers, of which 2 defects have been validated so far.

The performance of Metallicus indicates the feasibility of an automated test mining technique. The results show that such a tool is useful even for a language-agnostic testing process to effectively reveal defects. To the best of our knowledge, Metallicus is among the first automated tools that leverage function similarities from the perspective of test recommendation. However, this contribution also opens research opportunities to address the challenges that come with handling various scenarios that Metallicus does not currently support. Section 6.4 discusses these challenges.

## 4.4 Discussion

### 4.4.1 Threats to Validity

#### 4.4.1.1 Internal Validity

The labelling of the dataset, used in the empirical study and evaluation of Metallicus, to validate similar function pairs has been done manually. We have ignored the program implementation of functions in our study and have only analyzed the documentation presented in natural language. However, it is essential to have knowledge about technical terms that are specific to programming logic or the programming language. To avoid any threat pertaining to experimental bias, we have ensured that the four participants involved in the labelling task have adequate programming experience of at least two years in both Java and Python. Furthermore, to avoid an experimental bias while conducting the user study discussed in Section 4.3.3, that required interpreting a functionality from the source code to prepare a documentation, we have ensured that the 6 participants involved are non-authors and have a development experience of at least three years.

#### 4.4.1.2 External Validity

The empirical study presents the observations drawn for 31,716 queries sourced from 12 libraries in 2 languages and 3 themes. Furthermore, Metallicus has been assessed on library functions from a limited set of themes and languages. One possible threat could be to the generalization of the study and the tool's effectiveness.

We have chosen languages that support multiple paradigms and are commonly used in software projects. Hence, our findings may be beneficial to a significant fraction of the developers. To study the theme-aspect of the study, we have chosen 3 diverse themes capturing 4 libraries from each theme. With the presented scale and diversity in our study, while the statistics may vary across other themes and libraries, the trends drawn from it are likely to have overlaps with the findings from our study. It would be interesting to see how the numbers vary for libraries based on themes such as machine learning and game development. Despite the variation in the extent of similarities across different languages and themes, the suggested approach may nonetheless find use in library testing.

Metallicus has a modular architecture. The match extraction stage is language-agnostic, as it handles natural language text. The algorithms designed for test extraction, parameter mapping, and test adaptation are generic. As a result, extending support to other test frameworks, languages, and themes of libraries is feasible by adding the respective parsing support and the domain-specific embeddings. We leave incorporation of more diverse themes and languages for the future.

#### 4.4.1.3 Construct Validity

The objective of this work is to assess the feasibility and effectiveness of an automated technique that leverages similarities among libraries and their usefulness in testing. Hence, we have used common metrics of precision and recall to assess the performance of the match extraction framework. As defect revelation is the key objective behind testing, we have considered coverage and the number of defects exposed as metrics to evaluate the effectiveness of leveraging test suites across libraries.

### 4.4.2 Scope and Limitations

We scope Metallicus's contribution to recommending test inputs with a potential to reveal defects. Metallicus returns test suites containing the assertions to indicate expected output for a given program input. However, there is a need to address the test oracle problem, which makes for an interesting research domain in the context of the proposed test mining approach. Addressing the test oracle problem is hard pertaining to scenarios observed while evaluating Metallicus. These scenarios are listed as follows.

- *False assertions due to inclusion relation of the function pairs:* Consider a query function `ipv6(value)` that validates if an input string is an IP version 6 address. A match candidate for this function is `isValid(String inetAddress)`, which is a superset function that validates whether the input string is an IP address. Note that IP address includes both IPv4 and IPv6. A lot of tests associated with `isValid` may be relevant to test the query function. However, there will also be some irrelevant tests of type ipv4 that Metallicus would recommend which would result in assertion failures that aren't defects in the query function. Hence, not all assertion failures must be indicative of a defect. It is an open problem to refine test cases and address the test oracle problem in the context of inclusion relationship while mining tests.

- *Customized configuration parameters:* While evaluating Metallicus, we observed cases of similar functions that allowed additional configuration to make a functionality more specific. For instance, consider the query and matched functions, `email(value)` and `isValid(String email)` respectively, which validate if the input is an email address. The latter allows additional configuration setting as a constructor argument to allow or disallow email address with a local domain, with the default setting being to disallow address with local domain. In contrast, the query function does not have any such configuration. As a result, while in the recommended case, the expected output for a test input `‘joe@localhost’` is `False`, the query function expects the output to be `True`. This results in an assertion failure that does not reveal a defect.

- *Unadaptable output types and tricky type conversions:* Despite performing similar functionality, two functions may have different output expectations. While for some cases conversion to another type may be possible by following certain heuristics, in others, this may be non-trivial due to the incompatibility of the data types. Consider the previously discussed example of `email(value)` and `isValid(String` `email)` functions. While the former returns `True` for a valid email and throws a `ValidationFailure` for an invalid email, the latter returns the Java boolean types `true` or `false`. METALLICUS supports cross-language boolean conversion from `True` (in Python) to `true` (in Java). However, to convert from `false` to `ValidationFailure`, METALLICUS follows a heuristic-based approach, in which it maps an exception to a boolean `false` when there is an incompatibility of types. Such heuristics can't be generalized and at times such a conversion is non-trivial. For instance, consider a query function, `wrap(final String str, int wrapLength)`, and its matched function, `wrap(text, width=70)`; these functions wrap a given string at a given length. The first function returns a string wrapped using newline characters, while the second function returns a list of strings that represents the original string split at a wrap-length. For a fully automated approach, conversion from list type to single string is non-trivial, especially when the delimiter to concatenate the string elements in the list is unknown or non-specific (newline in this case). Such scenarios may result in falsely introduced assertions. Nonetheless, as the main functionality is the same, the inputs make relevant test cases and can not be overlooked. There has been work on automated adaptation between object data types that share semantics, such as arrays and Java Collections Framework [127]. However, for the presented example, determining the semantic similarity between String and List types is itself a challenge and an open problem.

## 4.5 Conclusion

We presented a test mining approach that leverages similarity among functions across different libraries and their associated test suites to reveal defects in one another. We followed a two-step approach in which we first conducted an empirical study to assess the idea, which was followed by fully automating the idea into a tool to implement the prototype. Our empirical evaluation indicated the existence of a significant number of similar functions across libraries in the same programming language as well as in different languages. Test suites from another library can serve as an effective source of defect revealing tests. The study resulted in the revelation of 67 defects across 12 libraries. We further validated the feasibility of the automation of this approach in the form of a test recommendation tool, METALLICUS. METALLICUS utilizes the textual data associated with functions to apply natural language processing in order to retrieve relevant match functions for a given query function. The test extraction module of METALLICUS implements a generic approach to extract the tests associated with the matched function by performing static analysis. METALLICUS further processes the tests to synthesize test suites for the query function. Evaluation of METALLICUS resulted in the revelation of 46 defects, of which 24 defects are previously unknown defects. The results show that METALLICUS is useful even for a language-agnostic testing process, to effectively reveal defects.

Apart from extending METALLICUS's support to more languages and themes in the future, we plan to work on the test oracle problem, pertaining to the challenges discussed in Section 4.4.2. One step, in this direction, would be to use the user's feedback report, containing execution traces of the test suite, to identify genuine and non-genuine assertion failures. Intuitively, large number of failures may be indicative of non-genuine assertion failures, which may be submitted to METALLICUS for further refining.

# Chapter 5

# Unveiling Semantic Inconsistencies between Code and Specification

## 5.1 Introduction and Motivation

Oftentimes, a programmer may implement a functionality that differs from specifications either in the documentation or in standard conventions followed for inputs, especially, structured inputs such as a file path or URL. These structured inputs, despite being valid as per the specification, may end up being handled incorrectly in the implementation if logical checks on the expected input are missing or weakly defined or if there are additional checks in the code that constrain the input as compared to its specification in the documentation. Qualitative evaluation of test cases based on code- or path-coverage ([26, 147, 27, 77, 155]) may overlook such issues.

Consider the defect #1462[1] in Google Guava library's *getFileExtension* method in *Files* class. According to the specification, as obtained from the javadoc, the method accepts a string denoting a file name and returns the file extension or an empty string if the file has no extension. The faulty implementation, shown in Listing 5.1, misses cases where a dot ('.'), the extension separator, may be present in the internal path components of the file name. For instance, the method with input `C:\abc.def\testfile` would incorrectly return the extension as `def\testfile` when it should return an empty string. The implementation returns the substring following the last '.'. However, the implementation misses that the file separator ('\' in this case) may occur after the dot, which is valid for a file path.

Appropriate constraints to extract a valid file extension are not explicitly provided either in the main code or as assertions by the developer. As a result, passing the program-path constraints to a constraint solver or using a fuzz tester is unlikely to generate tests to reveal the described defect.

We know from the documentation that there is only a single '.' as extension separator, while rest of the occurrences of '.' act as non-separators. The implementation presumes all occurrences of '.' to act as separators. Our proposed technique leverages this gap by using a static analysis on strings to formulate two regular expressions from the documentation and the implementation, respectively, to denote the input

This work appeared at the IEEE/ACM International Conference on Automated Software Engineering (ASE), November 2019

[1]`https://github.com/google/guava/issues/1462`

```java
/** Returns the file extension for the given file name, or the empty string if
    the file has no extension. The result does not include the '.'. **/
public static String getFileExtension(String fileName)
{ checkNotNull(fileName);
  int dotIndex = fileName.lastIndexOf('.');
  return (dotIndex == -1) ? "" : fileName.substring(dotIndex + 1); }
```

**Listing 5.1:** Defective implementation of getFileExtension.

structure treating separator '.' as a different character altogether from the non-separator '.'. Using this approach we generate the potentially defect exposing test strings that denote the differences in the two regular expressions.

Our work focuses on test generation of structured and semi-structured inputs commonly specified using string-based data types. The approach particularly targets cases of missing logical checks in the implementation or missing details in the specifications that are otherwise part of the implementation. A solution to deal with such inconsistencies may be specific to the nature of input data type. The situation becomes complicated with string inputs where the size of character-set (charset) may be large and the manipulation operations on strings may be diverse. These challenges make testing a non-trivial task. While researchers have done substantial work on detecting input validation defects in a method ([82, 156, 126, 11]), these techniques may not necessarily expose subtle defects in the method once the input passes all validation constraint checks.

As string manipulation could be a source of vulnerabilities in security critical programs [109, 83, 131, 157, 158, 13], string analysis has been actively researched upon in the last two decades [132, 64]. Christensen *et al.* present a precise string analysis which statically extracts flow graph from string operations in a program to infer the structure of the derived string in the form of a context free grammar [32]. The technique was shown to be effective in validating syntax of standard inputs like SQL queries. However, to test a client program the approach is dependent on the existence of an input validator program, which may not be present in all cases. A documentation is more likely to be available instead. By generating the shortest possible counter-example violating the specified grammar, this technique limits the revelation of several potential defects for which a well defined string generation model may be required. Yu *et al.* propose a symbolic analysis approach that checks if the symbolic string reaching the sink of the program matches a given vulnerable pattern. A forward analysis learns manipulations of *concatenate* and *replace* functions on strings to model as automaton. Using forward reachability analysis they compute an over-approximation of all possible values that string variables can take at each program point and use the dependency graph to determine the flow of symbolic string [158]. Minamide proposes a static program analyzer that approximates the output of a program with a context free grammar and generates counter-examples using it, given specifications on the output to highlight potential vulnerabilities [93]. The string analysis proposed in the past has been predominantly done to derive the structural properties of the derived string. The analysis that we propose models implicit and explicit conditions not with the objective of capturing the modification to the string to learn the output structure but instead to learn existing constraints on the input string.

In this work we make the following key contributions:

1. A test generation technique to expose defects arising from string-based input inconsistencies between method implementation and specification. These inconsistencies may be semantic or syntactic in nature.

```java
/** Returns the username and password pair based on the specified encoded String
    obtained from the request's authorization header.
   Per RFC 2617, the default implementation first Base64 decodes the string and
   then splits the resulting decoded string into two based on the ":" character.
   param scheme: the authcScheme found in the request authzHeader. It is
   ignored by this implementation, but available to overriding implementations
   should they find it useful.
   param encoded: the Base64-encoded username:password value found after the
   scheme in the header.
   return the username (index 0)/password (index 1) pair obtained from the
   encoded header data.
**/
public String[] getPrincipalsAndCredentials(String scheme, String encoded) {
 String decoded=Base64.decodeToString(encoded);
 return decoded.split(":"); }
```

**Listing 5.2:** Defective getPrincipalsAndCredentials method.

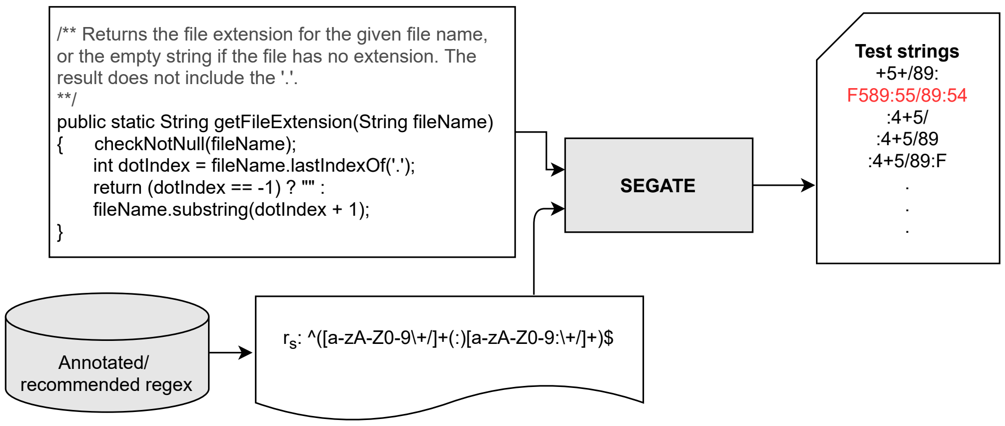
2. Design and implementation of a prototype tool, SEGATE, to assess the proposed technique on Java programs.
3. Exposing unreported defects in widely used libraries using the proposed technique.

SEGATE's novelty is in its differential approach that is combined with in-depth string analysis and prioritization heuristics to improve the quality of test strings. The technique allows targeting tests beyond the coverage criteria and input validation.

## 5.2 Overview

### 5.2.1 Motivating Example

Consider the defective implementation of *getPrincipalsAndCredentials* method, from Apache Shiro library, in Listing 5.2. The method documentation describes the *encoded* input as a Base64-encoded username:password value found in the request's authorization header. The method is expected to return the username and password pair split over ':'. RFC 2617, referred to in the documentation, states that the username excludes ':' character, but it is valid for a password to contain a ':', even though there is only a single ':' that acts as a separator. The implementation misses the support for this scenario, thus, depriving a user, having ':' in the password, from connecting to a Shiro protected application. For instance, for a Base64-encoded input *foo:this:bar*, the method returns password as *this*, at the second index of the returned String array, instead of the expected *this:bar*. The implementation has a single program path possible with the *split* function indicating the possible presence of ':' in the input. There is no additional check on single and multiple occurrences of ':', as both require a different way of handling. However, given a single constraint, it is not guaranteed for a constraint solver or a fuzzer to be able to generate concrete value containing more than one ':'. Thus, the defect, due to a missing constraint check, may be left unexposed.



```
/** Returns the file extension for the given file name,
or the empty string if the file has no extension. The
result does not include the '.'.
**/
public static String getFileExtension(String fileName)
{     checkNotNull(fileName);
      int dotIndex = fileName.lastIndexOf('.');
      return (dotIndex == -1) ? "" :
      fileName.substring(dotIndex + 1);
}
```

$r_s$: ^([a-zA-Z0-9\+/]+(:)[a-zA-Z0-9:\+/]+)$

```
Test strings
+5+/89:
F589:55/89:54
:4+5/
:4+5/89
:4+5/89:F
.
.
.
```

**Figure 12:** Working Prototype of SEGATE

Our technique models the structures of the specified input and the implementation-derived input as regular expressions (regexes), $r_s$ and $r_i$, respectively. These regexes may accept different languages. For the *encoded* input, $r_s$ is obtained by annotation, derived from the grammar in RFC 2617[2], as below

```
^([a-zA-Z0-9\+/]+(:)[a-zA-Z0-9\+/:]+)$
```

SEGATE statically analyses the method to derive $r_i$ as

```
^(([a-zA-Z0-9/:\+]*(:)[a-zA-Z0-9/:\+]*)+)$
```

As the colon (':') can act as a username-password separator or a non-separator, SEGATE assigns a different character representation in the regex as $(:_{nonspl})$ for the occurrence of colon as a non-separator in $r_s$. We generalize this idea to all characters that could play multiple roles in a string structure.

The approach models $r_s$ and $r_i$ as two separate Deterministic Finite Automata (DFA) to compute the differential DFA, $D_\triangle$. $D_\triangle$ represents the symmetric difference between sets of strings accepted by $r_s$ and $r_i$. SEGATE applies prioritization heuristics to generate a subset of test strings accepted by $D_\triangle$. For the discussed example, SEGATE successfully generates test strings with ':' occurring in the password of the encoded input, as described in the reported defect, hence, exposing the defect. Fig. 12 shows the input passed to and the output obtained from SEGATE for this particular example, indicating the defect revealing test string highlighted in red in the output from the tool.

### 5.2.2 SEGATE Overview

We implement the proposed technique in a prototype tool, SEGATE, Semantic Gap Tester. Fig. 13 shows the two main components of SEGATE, in grey boxes. Here is a flow of steps.

1. The Static Analyzer performs an inter-procedural dataflow analysis to compute the regex, $r_i$, denoting the input string to the method, as inferred by analyzing the program. The analyzer also extracts special characters from the program which are later used for test generation.

[2]https://tools.ietf.org/html/rfc2617#page-6

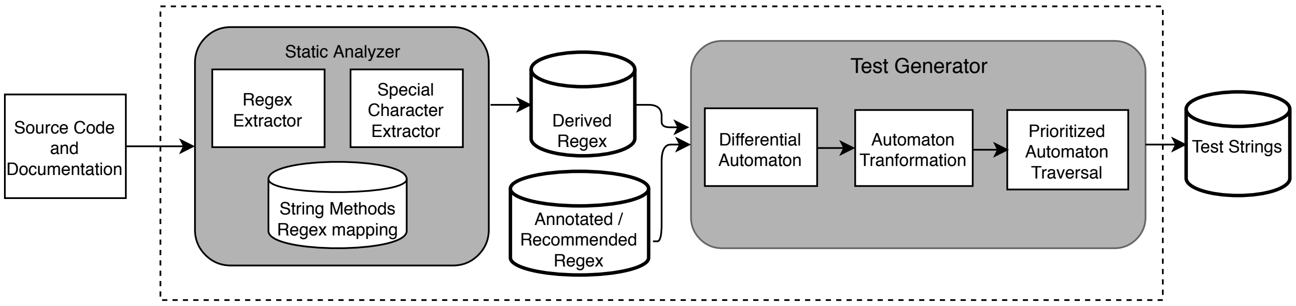

Source Code and Documentation
Static Analyzer
Regex Extractor
Special Character Extractor
String Methods Regex mapping
Derived Regex
Annotated / Recommended Regex
Test Generator
Differential Automaton
Automaton Tranformation
Prioritized Automaton Traversal
Test Strings

**Figure 13:** The architecture of SEGATE tool.

**Algorithm 1** Derive Regular Expression.

```
1: Input: M                                   ▷ Method to test in 3-address Jimple code
2: Input: RegMap                              ▷ regex mapping for all method inputs
3: function REGEX_FLOW(M)                     ▷ Initialize: Load regex mappings for String API
4:     RegMap := loadregexmapping()
5:     CG := M.getinterproceduralCallGraph()
6:     removeBackEdges(CG)                    ▷ remove cycles from CG
7:     methods := getmethods(CG)              ▷ Java lib. methods are ignored
8:     reverseTopologicalSort(methods)
9:     foreach $m \in methods$ do
10:        CFG := getUnitGraph(m)             ▷ control flow graph
11:        foreach $stmt \in CFG$ do
12:            compute InSet[stmt], OutSet[stmt]    ▷ Func. $6, 7$
13:        end for
14:        regex := OutSet[m.method_exit]     ▷ from Func. 7
15:        RegMap.add(m,regex)
16:    end for
17:    return RegMap
18: end function
```

2. $r_i$ is passed to the Test Generator, which also takes a human annotated or system recommended regex, $r_s$. Here $r_s$ denotes the set of input strings that are expected to be accepted by the method, as inferred from the documentation attached with the method being tested.

3. $r_s$ and $r_i$ are modeled as DFA from which SEGATE computes the symmetric difference between the two regexes to obtain a differential automaton, $D_\Delta$.

4. $D_\Delta$ is transformed into a graph. By performing a priority-based traversal on the graph, a set of test strings are generated that would be accepted by $D_\Delta$. These test strings represent the inconsistencies over input constraints in documentation and implementation.

We describe the components of SEGATE in the next section.

**Table 7:** Flow functions used in String Analysis (SA).

$$FlowSet : \{r \mid regex\, containing\, characters\, c \in ASCII\, charset\} \tag{3}$$

$$kill_{SA}(stmt) = \begin{cases} \emptyset & stmt \notin taintedStmt \\ all & otherwise \end{cases} \tag{4}$$

$$gen_{SA}(stmt) = \begin{cases} \emptyset & stmt \notin taintedStmt, \\ & \quad stmt = ThrowStmt \\ concatenate(e, \text{‘\&’}, r)\ \forall e \in InSet(stmt) & stmt \in \{IfStmt, LoopStmt, \\ \quad where\, r\, is\, regex\, obtained\, from\, RegMap^{*} & \quad InvokeStmt, AssgnStmt \\ & \quad with\, string\, operation\} \\ & \quad and\, stmt \in taintedStmt \\ \bigcup_{using\text{‘|’}}, \forall e \in InSet(stmt) & ReturnStmt \end{cases} \tag{5}$$

$$InSet(stmt) = \begin{cases} \{.*\} & at\, method\, entry \\ \bigcup\{OutSet(s') \mid s' \in pred(stmt)\} & otherwise \end{cases} \tag{6}$$

$$OutSet(stmt) = \begin{cases} \bigcup_{using\text{‘|’}}, \forall e \in InSet(stmt) & at\, method\, exit \\ InSet(stmt) \backslash kill_{SA}(stmt) \cup & otherwise \\ \quad gen_{SA}(stmt) & \end{cases} \tag{7}$$

**RegMap* contains mapping from method to the regex denoting the input structure derived from the implementation.

## 5.3 Tool Architecture

Fig. 13 shows the two main components of SEGATE, in grey boxes.

### 5.3.1 Static Analyzer

We apply static dataflow analysis to derive the regex, representing the structure of string input, from the source code. The Static Analyzer, built using Soot [76], performs forward-flow, inter-procedural and context-sensitive string analysis.

Algorithm 1 describes the derivation of regex from the method. The analysis takes the method under test in the form of a 3-address code in Jimple format [43] and returns the string representing the regex for each string input accepted by the target method. The `Regex_Flow` function obtains the call-graph to arrange methods in the reverse weak topological order [23] (Algo. 1, line 3-5) to perform an inter-procedural analysis and compute the analysis summary.

Using the flow functions in Table 7, the analyzer further performs string analysis over statements of each method (Algo. 1, line 9-14). In the rest of the chapter, we refer to a string-based parameter as input. The data flow facts (*FlowSet*) constitute of a set of regular expressions. The *InSet* at a method statement contains the regex status of the input as derived from the control flow paths reaching from all predecessors to that statement, denoted by *pred(stmt)* in function 6, condition 2 (func. 6, cond. 2). On processing the statement, the *InSet* elements are updated (based on $gen_{SA}$ and $kill_{SA}$ for the statement type) to obtain the *OutSet* (func. 7). Note that the presented string analysis is not a bit-vector analysis, and $gen_{SA}$ and $kill_{SA}$ depend on the *InSet*. The domain of the regular expressions is infinite, this would result in a lattice of infinite height. Section 5.3.1.2 discusses the heuristic we follow for the analysis to converge.

The analysis uses taint propagation to extract statements affected by the input and hence, contribute in generating regex. At the method entry, we assume the input parameter as tainted and add it to the inset. Further, using a *may* variant of constant propagation, the local variables referring to a constant string at an instance are replaced by their corresponding possible constant values. Our implementation of constant propagation also handles obtaining new string constants due to statements involving string concatenation functions and expressions. These constants are useful in building the regex.

#### 5.3.1.1 Summary Computation

`RegMap` maintains mapping (summary) from method to regex denoting the structure of the input derived from the method implementation. `RegMap` is updated with summaries as more methods are analyzed (Algo. 1, line 15). Initially, we maintain a manually obtained regex mapping for methods in Java's String API. For instance, a regex for *startsWith(str)* method over alphanumeric charset is ((<str>)\w*). The analyzer replaces <str> with all its possible values, obtained through constant propagation analysis, described previously. We assume that the third party library methods written in Java, dealing with string inputs, may at some point invoke the methods from Java library's String class. Hence, having these basic mappings would aid in building regex for string inputs of other methods. While obtaining this initial mapping, we ignored update-methods like 'concat' and its variants. These methods make additions to the tainted string without revealing any information about its original structure.

We stop exploring the depth of the call-graph at a method invocation from the Java library. The analyzer fetches the corresponding regex from `RegMap` for the invoked method.

#### 5.3.1.2 Statement Analysis

For a method, the `Regex_Flow` function generates regex by processing each taint-affected statement (*tainted-Stmt*) according to its type, such as conditionals, loop conditions, assignments, invocations and return.

For our analysis, we are only interested in a statement if it contains at least one tainted use-value. For instance, in an assignment statement, having a tainted use-value indicates that the variable being defined on the l-value is directly or indirectly affected by the input string and hence, would be useful to infer the structure of the input. In statements containing method invocations, if the implicit parameter of the invoked method is tainted, we pass the tainted values and constant arguments of the invoked method to obtain the corresponding regex from the existing mapping (in `RegMap`) of the invoked method. This regex is then appropriately concatenated with the existing regex in the inset according to the flow func. 5 (cond. 2) to obtain the outset (func. 7, cond. 2). Consider

*int dotIndex = fileName.lastIndexOf('.');*

From this statement one can infer that fileName may contain the character '.', hence, deriving regex [a-zA-Z0-9_\-\.\\]*(\.)?[^\.]* for this statement.

A typical conditional statement has a boolean expression which may be an (in-)equality check or a method invocation. We treat Switch-Case statement conditions and loops conditions similar to If-Else statement. Conditionals involving tainted values are relevant for our analysis to infer details about the input. Similar to the assignment statements, we maintain a condition-to-regex mapping, in `RegMap`, to build on the regex from the conditional statements. To reach a converging fixed point while analyzing statements inside a loop body, the derived regex from the loop body is not concatenated in the outset elements if already done in the previous iteration. If there is a new learning at each iteration of the analysis then a bound of three iterations was put.

Condition leading to a Throw statement that raises exception may not be useful in learning about the input as it indicates a case already handled for an invalid input. Hence, for every Throw statement, the analyzer kills all elements (func. 4, cond. 2) in the inset and generates nothing (func. 5, cond. 1). However, the analyzer still captures the negated condition that takes the 'else' path. The intuition is that the negated condition would define constraint on a 'valid' input.

For a return statement, the regex derived along all paths reaching it are concatenated using a '|' indicating logical *or* (func. 5, cond. 3). As all return statements lead to a method exit, the regex coming from all return statements are concatenated using a '|' indicating logical *or* (func. 7, cond. 1). The final single regex thus obtained, represents the set of strings that the input would match. This regex is added to `RegMap` corresponding to the method analyzed.

Statements of other types do not influence the regex and are hence, ignored.

#### 5.3.1.3 Handling multiple string inputs

If a method consists of multiple string parameters (inputs), then for every input at a time, we mark it as tainted and others as untainted to proceed computing its regex. This idea is similar to symbolic execution treating one input as symbolic and others as concrete.

#### 5.3.1.4 Collecting special characters

Any hard-coded character or string occurring in a statement is recorded in a set of special characters. This set is passed to the Test Generator. Section 5.3.2 discusses the use of this set in detail.

### 5.3.2 Test Generator

The Test Generator aims to generate concrete test inputs that are supported by exactly one of the two: the method implementation or the method specification obtained from the documentation. The regex $r_i$, derived by the Static Analyzer, is modeled into a Deterministic Finite Automaton (DFA), using the dk.brics.automaton library [97]. From the specifications, we infer and annotate the regex, $r_s$, for the string input, which is further modeled into another DFA.

#### 5.3.2.1 Obtaining annotations from specification

Similar to past techniques that take assertions or contracts as inputs (such as with Randoop [106]), SEGATE takes $r_s$ annotations from an expert or as recommendation from our system. To obtain $r_s$, we follow a semi-automated approach using an annotation recommender system. We build a repository of regex annotations for standard string structures to recommend annotations to further test other methods taking similar inputs. The repository contains a tag associated with every annotation, describing what the annotation denotes, such as 'json string', 'ip address', etc. To obtain recommendation for $r_s$, the system takes a query description as input and computes a phrase similarity score between the query and every tag in the repository, picking the tag with the highest score (ranging between 0 and 1), provided the score exceeds the empirically determined threshold of 0.55. To compute the similarity score, we use a semantic similarity computation API ([85]). The regex corresponding to the picked tag is recommended. If no appropriate tag exists, the user provides an annotation, which is fed into the system, along with a provided tag, for future recommendations. Such an approach thus reduces manual efforts and allows incorporating domain knowledge about the input structures, which could be leveraged to test other methods.

#### 5.3.2.2 Obtaining a differential DFA

The two DFAs, modeled from $r_i$ and $r_s$, are minimized using Hopcroft's algorithm [66]. SEGATE computes a differential DFA, $D_\Delta$, accepting the set of strings which is the symmetric difference represented by:

$$(r_s \cup r_i) - (r_s \cap r_i) \tag{8}$$

Using a DFA for string generation provides flexibility as well as control to generate diverse strings by traversing different paths of the automaton. Automaton model could also be used to generate strings by traversing selective paths.

***Need for differential automaton:*** The differential nature of $D_\Delta$ makes the strings accepted by it as relevant test inputs. We expect most $r_i$ complying strings to comply with $r_s$, and vice versa. Hence, we use a differential approach to narrow down to potentially defect-revealing strings.

#### 5.3.2.3 DFA transformation for string generation

The ASCII charset and the special characters obtained from the Static Analyzer make up the alphabet for the DFA. However, $D_\Delta$ may accept a large number of strings. To prepare test inputs, the requirement is to obtain a smaller subset of strings accepted by $D_\Delta$. Thus, the generator generates a subset of strings such that every sequence of transitions from each start to accept state in $D_\Delta$ is traversed not more than once. To ensure that every transition is visited once, we transform $D_\Delta$ to $D_t$, treating each transition as a *Node* and each pair of adjacent transitions as an *Edge*. Fig. 14 shows an example of a snippet of $D_\Delta$ being transformed to produce $D_t$. Algorithm 2 describes the derivation of $D_\Delta$ (line 4-6) and its transformation to DFA $D_t(Q, \Sigma, \delta, q_0, F)$ (line 7-24), defined as below:

- Q: Set of finite Nodes formed where a pair of adjacent States of $D_\Delta$ form a Node, also called a State Pair Node
- $\Sigma$: Alphabet containing the ASCII charset and the special characters' set obtained from the Static Analyzer
- $\delta$: Set of transitions between all adjacent State Pair Nodes with the transition function $\mathrm{Q} \times \Sigma \rightarrow \mathrm{Q}$. For adjacent State Pairs Nodes N1 and N2, N1.secondState = N2.firstState
- $q_0$: Start Nodes which are the set of nodes with the first State as a start State of $D_\Delta$
- F: Subset of Q containing accept Nodes denoting strings accepted by the regular language of expression 8. A Node N is in accept state if N.secondState is an accept State.

**Algorithm 2** Derive $D_\Delta$ and $D_t$.

```
 1: Inputs: Regex Strings $r_i$, $r_s$
 2: Output: Transformed Automaton $D_t$
 3: function TRANSFORM_AUTOMATON($r_i$, $r_s$)
 4:     Automaton $a_i$ := $r_i$.toAutomaton()
 5:     Automaton $a_s$ := $r_s$.toAutomaton()
 6:     Automaton $D_\Delta$ := Difference(Union($a_i$, $a_s$), Intersection($a_i$, $a_s$))
 7:     foreach Transition StatePair (s1, s2) $\in D_\Delta$ do                ▷ Transformation
 8:         CharSet symbolset := partition((s1, s2).symbols, splchars)
 9:         foreach $sym \in symbolset$ do
10:             Node n := Node((s1, s2), sym)
11:             if s2 is acceptState then
12:                 n.setAcceptNode
13:             end if
14:             if s1 is initState then
15:                 n.setInitNode
16:             end if
17:             if s1 = s2 then
18:                 n.setSelfLoopNode(true)
19:             end if
20:         end for
21:     end for
22:     foreach adjacent Nodes n1, n2 do
23:         Edge e := Edge(n1, n2)                ▷ where n1.secondState=n2.firstState
24:     end for
25:     return Automaton(Set<Node>, ASCIICharSet, Set<Edge>, Set<InitNode>,
        Set<AcceptNode>)
26: end function
```

#### 5.3.2.4 Test string generation from $D_t$

To obtain the test strings, we apply a variant of depth-first search traversal (`DFS_Visit` function of Algorithm 3) over $D_t$ to visit every node, that represents a transition edge from $D_\Delta$. The number of nodes in the automaton, despite minimization, ranged to large numbers, with maximum 6866 nodes for a method in our dataset. Hence, while traversing $D_t$, we do not visit a node more than once, hence ignoring the respective path (Algo. 3, line 18). This optimization allows the technique to scale by avoiding path explosion and allows generating a smaller set of strings suitable for a test set. While traversing from every node, a symbol is randomly picked from the range of symbols supported for the node and concatenated to the string being derived (Algo. 3, line 19). On reaching an accept state node, the resultant string is logged and the traversal continues deriving other test strings (Algo. 3, line 21-24). For the *getPrincipalsAndCredentials* method (section 5.2.1), the corresponding $D_t$, due to its differential nature, would contain at least one path that accepts strings containing both the colons (: and $:_{nonspl}$) in the username:password encoding.

***Advantage of $D_t$ over $D_\Delta$:*** Note how the traversal over $D_t$ instead of $D_\Delta$ guarantees visiting every vertex of the original differential automaton $D_\Delta$. Referring to Fig. 14, to explain through an example, a

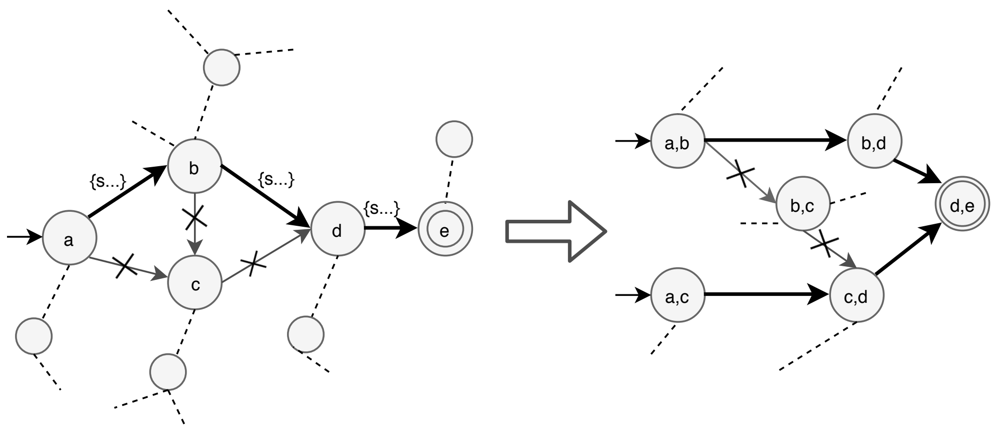


**Figure 14:** Transformation from $D_\Delta$ to $D_t$.

traversal along the path `a,b,d,e` on $D_\Delta$ would mark all the States (vertices) along the path as visited. This eliminates the paths `a,b,c,d,e` and `a,c,d,e` due to `b` and `d` already marked visited, leaving vertex c unexplored. Transformation to $D_t$, as shown in the figure, defines distinct nodes for every transition and running a depth-first traversal on this automaton would allow exploring more paths allowing every State (from $D_\Delta$) to be visited. Note how the path `a,c,d,e` (or `a,b,c,d,e`) could now be included in the traversal, in addition to `a,b,d,e`.

#### 5.3.2.5 Separate representation for special characters

Recall that the Static Analyzer derives special characters that are passed to the Test Generator. These characters appear directly in the implementation. Hence, they should be given more importance over other characters in the charset, while generating the test strings. In $D_\Delta$ the transition symbols contain a range of symbols merged into a single transition. These symbols may include special characters, SEGATE represents each of which on a separate transition edge. To introduce these separate transitions the *partition* function (Algo. 2, line 8) performs splitting of the symbol range. For instance, a transition with symbol range *a - g*, having *d* as a special character, would be split into three transitions with respective symbol ranges as *a - c*, *d* and *e - g*. Analogous to splitting the transition, $D_t$ would introduce additional State Pair Node with corresponding symbols set according to the separated transition edge. On traversing $D_t$, these special character edges are given higher priority over other edges (Algo. 3, line 16). Doing so provides the advantage of generating strings containing special characters.

***Design decisions and trade-offs:*** The Static Analyzer assumes that if a variable is tainted, then the constraints on it are relevant to infer $r_i$ of the input that tainted it. For instance, in Listing 5.2 from Section 5.2.1, *decoded* is tainted by the input *encoded* as the inter-procedural analysis results in *decodeToString* invocation returning a tainted output. Hence, the taint is propagated on to *decoded*. As a result, the subsequent call on *decoded.split(":")* induces the constraint of ":" being present in *decoded* as well as *encoded*. The constraint on *encoded* is valid for the given example, as inferred by studying the nature of manipulation on *encoded* by *decodeToString*. However, in general, this propagation of constraint on the actual input may not always result in an accurate inference. The analyzer could possibly relax the constraint on the input into a ‘possible’ occurrence instead of ‘strict’ occurrence. However, this relaxation would increase the exploration space of strings. Due to scalability consideration, the analyzer imposes strictness on such constraints.

**Algorithm 3** Generate Test Strings.

```
1: Input: $D_t$
2: Output: String[] testStrings
3: function GENERATE_TESTSTRING($D_t$)
4:     foreach initState src $\in D_t$ do
5:         teststrings.add(DFS_Visit(src, visited:=[], $D_t$, str := "", teststrings := []))
6:     end for
7:     return teststrings
8: end function
9: Input: srcNode, visitedNodes, $D_t$
10: Output: String[] testStr                                  ▷ list of test strings
11: function DFS_VISIT(src, visited[], $D_t$, str, testStr[])
12:     visited.add(src)
13:     if $src.isAcceptNode \wedge str \notin testStr$ then
14:         testStr.add(str)
15:     end if
16:     prioritize(src.neighbors)
17:     foreach neighbor of src do
18:         if $neighbor \notin visited$ then   ▷ concatenate random char from the symbols of the
    Node
19:             str := str.concat(neighbor.RandomSymbol())
20:         end if
21:         if neighbor.isAcceptNode then
22:             testStr.add(src)
23:         end if
24:         DFS_Visit(neighbor, visited, $D_t$, str, testStr)
25:     end for
26:     return testStr
27: end function
```

Due to the size of the differential automaton being large, capturing every transition edge would not scale. As a result, we applied the transformation to $D_t$ to allow covering equal or relatively more number of paths of string sequences than the number we could have covered on $D_\Delta$. To generate a reasonable number of test strings, we ignore paths on which the nodes have been visited before. While this allows the approach to scale and results in a set of test strings representing the gap, it comes with a trade-off on missing certain interesting paths from the perspective of producing defect revealing strings. Improving on the traversal heuristics would be of our interest.

## 5.4 Evaluation and Results

We evaluated SEGATE on its ability to expose defects in a method or its specification. We ran SEGATE: i) on methods with known defects to verify if it generated test inputs to expose the defects ii) on methods in popular libraries with the objective to expose potential defects iii) to assess its performance relative to

**Table 8:** List of some of the known defects from popular libraries used for evaluation.

| BugId | MethodName | Library | Defect Description |
|---|---|---|---|
| IO-499 | FilenameUtils. directoryContains | Apache Commons-IO | Gives false positive when two directories have equal prefixes |
| IO-567 | FilenameUtils.getExtension | Apache Commons-IO | Implementation misses accounting for existence of alternate data stream in file-path |
| http-392 | UrlEncodedParser.parse | google-http-java-client | A parameter with '=' in the value is misinterpreted |

state-of-the-art techniques. All evaluations were performed on a Windows 10 64-bit system with Intel Core i7-5500U 2.40GHz processor with 8GB RAM and maximum JVM heap-size of 4GB. SEGATE has been built over Eclipse Mars 4.5.2 IDE running with Java 1.8. The dataset and results are available at the project website.[3]

### 5.4.1 Verifying against known defects

#### 5.4.1.1 Methodology

We picked commonly used string structures, often used in research [129], to evaluate program methods taking such structured input. These structures include file path, URL, JSON format, IP address, phone number, email and credentials. To prepare a benchmark dataset with known defects, we extracted top 30 starred Java projects on GitHub. Filtering on keyword search over these repositories on the structure names mentioned, we obtained 14 projects that had at least one method taking at least one such structure as input. We inspected the issue trackers of these 14 projects and extracted issues labeled as defect. We considered defects no older than reported in the year 2012, to avoid the scenario where the method of concern is obsolete. We further picked 1 more project which has been frequently referred to in some of these 14 projects and followed the same filtering approach on it. This resulted in analyzing issues in 15 open-source projects. From the filtered issues, we shortlisted defects discussing methods that take at least one parameter as a string-based structured input and discuss one or more of the following cases: i) there is a difference in what the documentation or the referenced specifications state and the code behavior; ii) there is a practical use-case that is not handled well by the implementation. Some such defects are listed in Table 8.

We obtained a dataset of 34 such defects in distinct methods, across 15 libraries. Corresponding to every method were Java documentation or external documentation, describing the input structure. If our recommender system lacked a relevant regex annotation ($r_s$), we assigned an annotator, a computer science graduate having a development experience of 2 years in Java, with the task of deriving $r_s$ from the documentation. The annotator was provided with the documentation of the faulty method and the method parameter for which we require obtaining $r_s$. To avoid bias, the annotator was not informed about the purpose of the study. One of the authors performed the same task to validate the annotations derived. For 85% methods in our dataset, the documentation referred to external sources, such as RFCs, that contained

[3]`https://github.com/pag-iiitd/Segate`

**Table 9:** Evaluation Data Description.

| **Data description** | **Value** | |
|---|---|---|
| # Faulty methods analyzed | 34 | |
| Code size (LOC) | Mean: 495 | Stdev.: 676.9 |
| Range of code size (LOC) | 14 - 2955 | |
| Call-graph depth | Mean: 31 | Stdev.: 44.6 |
| Range of Call Graph size | 1 - 176 | |

**Table 10:** Performance of Segate in exposing defects.

| **Metric** | **Value** |
|---|---|
| # Known defects in dataset | 34 |
| # New defects exposed using Segate | 6 |
| # Known defects exposed by Segate | 26 |
| # Total defects exposed by Segate | 32 (of **40**) |
| # Test strings generated per run | Mean: 1346 Stdev.: 2894 |
| Range of # of test strings generated | 2 - 12266 |
| Execution time(s) per run | Mean: 8.23 Stdev:. 9.05 |
| Memory consumption(MB) per run | Mean: 60.89 Stdev.: 15.58 |

grammar defining the syntax of the string input. The annotators were required to derive $r_s$ from the grammar. A manual annotation on an average required 18 minutes. This exercise was required for 19 of the 34 methods. For the remaining 15 methods, based on our domain knowledge, we felt that the recommendation for $r_s$, returned by our system, is fit for usage. To assess the inter-rater agreement, the free-marginal kappa score [113] was computed as 0.68, which is rated as 'fair to good' [1]. In 3 of the 19 annotations, there was a minor disagreement among annotators, which was resolved over discussion.

The final annotation, $r_s$, was then fed to Segate to generate test strings complying with expression 8. These annotations were added in the recommendation repository for future query. As the repository expanded, we observed improved recommendations. This indicates the usefulness of manual intervention, in annotating, as the domain-knowledge gained from experienced developers strengthen the recommendations.

We ran Segate on the faulty methods in our dataset to check if it generated test inputs to expose the defects. We assess Segate on the number of defects exposed by its tests, and its execution time and memory usage. Table 9 gives a statistical description of the nature of programs in our dataset. The description includes the range and the average size of the code (LOC), over 34 faulty methods. The code size is a count on the number of distinct statements (including inter-procedural method body), in Jimple representation, visited by the Static Analyzer. The call-graph depth refers to the depth of method invocations involved in the method under analysis.

#### 5.4.1.2 Results

Segate generated test strings exposing 26 of the 34 known defects. Table 10 gives the details of the execution time and memory consumption of Segate, with values averaged over 10 executions over every method. Over each program, Segate executed in a matter of a few seconds, with the mean value of 8.2 seconds.

```
1  while(index < json.length())
2  { char current = json.charAt(index);
3    if (current == '{')  inObject++;
4    if (current == '}')  inObject--;
5    if (current == '[')  inList++;
6    if (current == ']')  inList--;
7    if (current == ',' && inObject == 0 && inList == 0) {
8      list.add(build.toString());
9      build.setLength(0); }
10   else   build.append(current);
11   index++; }
```

**Listing 5.3:** Snippet indirectly manipulating occurrence of comma.

Discounting the memory consumed by the garbage collector, SEGATE's memory consumption on execution ranged between 43 to 106 MB, as evaluated on every method in the dataset.

For the 8 defects missed by SEGATE, following are the key observations to infer the reason for failure.

i. *Indirect manipulation on position of characters in the string*: Consider the faulty method reported in the defect, Spring-Boot-11992, which constrains the occurrence of comma (',') relative to '[', ']', '{' and '}' (statement 7 of Listing 5.3). This constraint is enforced by integral counters. The Static Analyzer does not currently support making such precise inferences from numeric manipulation on strings. The analyzer infers the occurrence of comma (',') anywhere in the string. Hence, SEGATE misses exposing the defect. However, the analyzer has basic support for inference of regex from character position. For instance, *str.charAt(2)==':'* results in a regex indicating ':' at third position in *str*. Index-based manipulation on substrings is handled based on certain templates to infer the position as either at front, last or in between the string.

ii. *Missing explicit checks from the specification as well as the implementation*: The case reported in VALIDATOR-420 is about a validation method incorrectly accepting url containing a space character. As both, the specification and the implementation, do not have an explicit constraint barring occurrence of a space in a url, SEGATE misses exposing the defect. As the charset of invalid characters may be huge, we do not currently consider occurrence of a character, while modeling $r_i$ and $r_s$, if it does not explicitly appear in the implementation or the specification. We would be interested in obtaining implicit characters that are common sources of errors, in the future. Nonetheless, adding the space (' ') in the charset of $r_s$ allowed SEGATE to reveal the defect.

### 5.4.2 Exposing unreported defects

#### 5.4.2.1 Methodology

With an objective to explore new defects, we picked the first 20 test strings generated (or all if # test strings $< 20$) for each of the faulty methods obtained in Section 5.4.1 and passed them as inputs to the respective method and logged the output. The potential defects may not necessarily lead to an exception throw; the program could silently give an unexpected output. Hence, we manually look out for unexpected output, indicative of an exposed defect, based on the intended behavior specified in the documentation. Our work contributes in generation of quality test inputs to help a tester prioritize the tests.

**Table 11:** List of unknown defects from popular libraries exposed by SEGATE.

| No. | MethodName | Library | Defect Description |
|---|---|---|---|
| 1. | Files. getFileExtension | Google Guava | The method does not handle ADS file paths supported in NTFS |
| 2. | InetAddresses. isInetAddress | Google Guava | Gives false positive for ipv6 address in certain cases of occurrence of 0s such as in the last bit group for 804C:404C::0:CC:C0:00C0C |
| 3. | JsonParser. parse | Spring-Boot | Several validation checks missing in JsonParser- no check on closing '}' |
| 4. | PhoneNumberUtil. parse | libphone-number | Specifications from the RFC reference in the documentation are loose as compared to the stricter checks in the actual implementation |
| 5. | UriUtils. extractFileExtension | Spring | Validation check missing on the url path |
| 6. | InetAddressValidator. isValidInet6Address | Apache Validator | Does not account for presence of scope id in the IP address |

#### 5.4.2.2 Results

By manual inspection, we could spot 4 new defects in the methods in our dataset (Table 11). On increasing the limit to inspecting first 150 test strings for each method, we explored 2 more new defects. We have reported these defects to the developers and have so far received validation of four defects.

### 5.4.3 Comparative evaluation of SEGATE

#### 5.4.3.1 Undirected Fuzz Tester

To compare our technique with black-box undirected fuzzing, we implemented such a fuzzer using a semi-randomized string generation approach. To generate meaningful strings, the fuzzer derives automaton from the annotated $r_s$ for every method in our dataset and generates test strings accepted by the automaton using depth-first approach, traversing random paths. Using $r_s$ in this approach makes the fuzzer as purely specification-based. For each method, the fuzzer generated same number of test strings as by SEGATE. Due to the randomized nature of the undirected fuzzer we computed each result over 10 program executions. We considered defect detection as successful if the fault revealing string was generated in at least 5 of 10 executions.

**Results** Table 12 gives the comparative summary of SEGATE and the undirected fuzzer. Column *'Gen.'* denotes the status of generation of defect-revealing string by the respective technique. On a set of 40 defects, while SEGATE exposed 32 defects, the undirected fuzzer exposed 16 defects. Of these 16 cases, for 2 cases the undirected fuzzer generated the fault revealing string for only 6 and 7 of the 10 executions, respectively, indicating unreliability of the technique. On the other hand, SEGATE uses a deterministic approach.

The undirected fuzzing technique missed the defect when the defect-revealing input had one of the following properties.

**Table 12:** Performance comparison of SEGATE with Undirected fuzzer and EvoSuite in generating fault exposing inputs.

| No. | BugId | SEGATE | | | Undirected Fuzzer | | | EvoSuite | |
|---|---|---|---|---|---|---|---|---|---|
| | | **# Test Inputs** | **Gen.** | **At** | **# Test Inputs** | **Gen.** | **At** | **# Test Inputs** | **Gen.** |
| 1. | async-http-client-1071 | 7217 | × | - | 7217 | × | - | 45 | ✓ |
| 2. | async-http-client-1110 | 3439 | ✓ | 1 | 3439 | ✓ | 1 | 156 | ✓ |
| 3. | async-http-client-1455 | 2368 | × | - | 2368 | × | - | 156 | × |
| 4. | GreenMail-213 | 1638 | ✓ | 2 | 1638 | × | - | 7 | × |
| 5. | Guava-1462 | 203 | ✓ | 12 | 203 | ✓ | 125 | 8 | × |
| 6. | Guava-1557 | 1093 | ✓ | 1 | 1093 | ✓ | 8 | 10 | × |
| 7. | Guava Files* | 203 | ✓ | 2 | 203 | ✓ | 14 | 8 | × |
| 8. | Guava InetAddresses* | 1042 | ✓ | 119 | 1042 | × | - | 10 | × |
| 9. | http-392 | 91 | ✓ | 4 | 91 | ✓ | 3 | 17 | × |
| 10. | IO-483 | 70 | ✓ | 1 | 70 | × | - | 31 | ✓ |
| 11. | IO-499 | 8 | ✓ | 2 | 8 | × | - | 20 | × |
| 12. | IO-545 | 243 | ✓ | 176 | 243 | ✓ | 244 | 27 | ✓ |
| 13. | IO-552 | 133 | ✓ | 1 | 133 | × | - | 30 | ✓ |
| 14. | IO-559 | 95 | ✓ | 64 | 95 | × | - | 39 | × |
| 15. | IO-567 | 195 | ✓ | 9 | 195 | ✓ | 25 | 27 | × |
| 16. | LANG-1374 | 3 | ✓ | 1 | 3 | ✓ | 1 | 1 | × |
| 17. | LANG-1395 | 239 | ✓ | 1 | 239 | ✓ | 1 | 147 | × |
| 18. | libphonenumber-1672 | 12266 | × | - | 12266 | × | - | 12 | × |
| 19. | libphonenumber PhoneNumberUtil* | 12266 | ✓ | 1 | 12266 | × | - | 12 | × |
| 20. | NET-582 | 202 | ✓ | 1 | 202 | × | - | 10 | ✓ |
| 21. | okhttp-2202 | 135 | ✓ | 1 | 135 | × | - | 8 | × |
| 22. | okhttp-2549 | 127 | ✓ | 8 | 127 | × | - | 10 | × |
| 23. | okhttp-2842 | 228 | ✓ | 8 | 228 | × | - | 8 | ✓ |
| 24. | okhttp-2939 | 9679 | ✓ | 181 | 10 | × | - | 10 | ✓ |
| 25. | SHIRO-375 | 95 | ✓ | 2 | 95 | ✓ | 23 | 24 | × |
| 26. | Spring UriUtils* | 36 | ✓ | 3 | 36 | × | - | 18 | × |
| 27. | Spring-15786 | 36 | × | - | 36 | × | - | 18 | × |
| 28. | Spring-Boot-3273 | 56 | ✓ | 3 | 56 | ✓ | 44 | 25 | × |
| 29. | Spring-Boot-6121 | 62 | ✓ | 13 | 62 | ✓ | 31 | 0** | × |
| 30. | Spring-Boot-12325 | 4451 | ✓ | 5 | 4451 | ✓ | 75 | 0** | × |
| 31. | Spring-Boot-12297 | 152 | ✓ | 1 | 152 | × | - | 115 | × |
| 32. | Spring-Boot-11992 | 152 | × | - | 152 | × | - | 130 | × |
| 33. | Spring-Boot JsonParser* | 152 | ✓ | 78 | 152 | × | - | 130 | ✓ |
| 34. | TEXT-118 | 2 | ✓ | 1 | 2 | ✓ | 1 | 10 | ✓ |
| 35. | UrlBuilder-5 | 270 | ✓ | 1 | 270 | ✓ | 5 | 214 | × |
| 36. | UrlBuilder-39 | 214 | ✓ | 7 | 214 | × | - | 8 | × |
| 37. | VALIDATOR-411 | 180 | × | - | 180 | × | - | 11 | × |
| 38. | VALIDATOR-419 | 254 | × | - | 254 | × | - | 62 | × |
| 39. | VALIDATOR-420 | 180 | × | - | 180 | × | - | 11 | × |
| 40. | VALIDATOR InetAddressValidator* | 254 | ✓ | 2 | 254 | ✓ | 23 | 62 | × |
| | **# Faults Exposed** | | **32** | | | **16** | | | **10** |

* indicates new defect exposed by SEGATE ** indicates defective method was not covered by EvoSuite

i. The input does not comply with the specifications but it is still accepted by the implementation due to missing or weak validation checks in the code.

For the above case, the defects could not be exposed by the generated strings that already comply with the specifications. Thus, considering the implementation becomes necessary.

ii. We also observed defects in which the defect-revealing string complied with the specification but it was missed by black-box undirected fuzzing. This happens due to the large space of compliant strings, reducing the probability of generation of tests having the defect-revealing nature. SEGATE improves the probability of generation of tests revealing unhandled boundary cases by eliminating trivial tests by using the differential approach (section 5.3.2). In our dataset, we did not find any defect exposed by the undirected fuzzer which SEGATE was unable to expose.

For a tester, it may be infeasible to evaluate a method on a large test set. Hence, we picked the first 20 test strings generated by both the techniques to observe if the defect-revealing strings appear in the set. Column *At* in Table 12 denotes the position of occurrence of the defect-revealing string in the test set returned by the respective technique for a method. The computed value has been averaged over positions obtained from 10 executions for the undirected fuzzing.

Using SEGATE, for 27 of the 32 successfully exposed defects, the defect-revealing string appeared in first 20 tests. Further, for 25 of the 27 defects, the relevant test string appeared in first 10 tests. This observation is attributed to the priority given to edges denoting special characters while traversing the differential automaton to generate test strings. For 5 cases where the relevant string did not appear in top 20, it was observed that leading strings contained other special characters (other than those specific to the defect in question) with equal priority, thus, pushing the defect-revealing string to lower position. Assigning priority among special characters could be subjective to the nature of the defect, hence, we leave developing heuristics for doing so for the future.

Using the undirected fuzzer, of the 16 successfully revealed defects, for 8 cases the relevant test string appeared in the first 20 tests, on an average. Further, for 7 of those 8 cases, the defect-revealing test string appeared in first 10 tests. For these 7 cases, we observed the nature of the relevant test string which in all the cases required occurrence of one or more special characters. The construct of the associated $r_s$ allowed occurrence of the special character(s) in high probability, resulting in a higher chance of defect-revealing string being generated by undirected fuzzing. For instance, for a regex ^[%/][0-9A-Fa-f]+$ there is a high chance of the percentage ('%') appearing in a string generated by undirected fuzzing.

### 5.4.3.2 Feedback-directed Fuzz Tester

Randoop [106] generates test-suites using a feedback-directed approach to iteratively build an input. Randoop uses feedback, from the execution of legal method sequences, to guide the search towards a new sequence resulting in new object states. We executed Randoop-4.1.0 ten times, using the default configuration, on every faulty method in our dataset. We have set the default time-limit of 100 seconds on the execution of Randoop, which is significantly higher than SEGATE's average execution time of 8.2 seconds observed on the dataset. However, Randoop did not reveal any defect in our dataset. On closely studying the generated test-suites in *ErrorTest* and *RegressionTest* files, we observed that most of the generated test inputs were invalid string-structures. To obtain meaningful tests, we passed a specification file to Randoop, describing the pre-condition that the input matches the corresponding $r_s$. However, Randoop appears to check the pre-condition as a filter after generating the tests, hence, not resulting in any useful test input.

**Table 13:** Configuration settings* used in tools.

| Tool | Parameter | Value |
|---|---|---|
| EvoSuite | Dminimize | false |
| JPF | symbolic.dp | choco |
|  | symbolic.string_dp | automata, sat |
|  | listener | gov.nasa.jpf.symbc.sequences.SymbolicSequenceListener |

*default values have been used for parameters other than those mentioned.

### 5.4.3.3 Coverage-based testing

To assess coverage-based techniques to test for semantic gaps, we executed Symbolic Path-Finder (SPF) [107] and EvoSuite [49] on our dataset.

**SPF** SPF uses the analysis engine of Java Path Finder (JPF), which is a model checking tool for Java programs. SPF combines path-coverage based symbolic execution technique with model checking [151] and it is known to have symbolic string support. We used SPF to verify the generation of test strings exposing defects in the methods from our dataset. Table 13 gives JPF configuration used in evaluation.

***Result:*** For only one method SPF generated a defect-exposing test string (BugId IO-552). Major reasons for missing the rest of the defects are listed as follows.

i. Use of native method calls is not supported by JPF [25].

ii. The path constraints are loosely defined in the method, capturing a wide variety of strings. While the solver produces a finite set of strings for every path constraint, it misses generating a string of defect-exposing nature.

iii. Explicit constraints are missing from the implementation and hence, are not captured by the path traversal approach used by symbolic execution.

To handle the last two cases, SEGATE leverages the information-rich specification in addition to the implementation. This allows elimination of redundant tests satisfied by both- the implementation and the specification.

**EvoSuite** EvoSuite generates test-suites using a search-based approach which optimizes on coverage criterion. We executed EvoSuite-1.0.6, on every class containing the defective methods from our dataset. While configuring EvoSuite, we disabled the test-set minimizer (to prevent exclusion of any code statement from analysis) and used the default coverage criterion that includes combination of criteria such as line coverage, branch coverage and method coverage. Due to the randomized nature of EvoSuite, the readings have been reported on 10 executions of EvoSuite for each defective method. We considered a defect as revealed by EvoSuite if at least one of the ten executions generated a defect revealing test string.

***Result:*** Of the 40 defects in our dataset, EvoSuite revealed 10 defects (Table 12), with mean coverage of 77.4% over the 40 cases. Of these 10 defects, one defect (BugId async-http-client-1071) was not revealed by SEGATE due to over-approximation resulting from confluence operation involved in the dataflow analysis. Hence, the coverage based technique was found to be more relevant to test for such a defect.

## 5.5 Conclusion and Future Work

We presented a test generation technique to expose defects in a method occurring due to input inconsistencies in the method documentation and the implementation. Focusing on structured string inputs, we used static analysis to model input as regex and comparing with the regex model derived from documentation, we computed strings to expose the gap between the two models. We evaluated our technique's implementation in our tool, SEGATE, which exposed defects in popular libraries. To assess the effectiveness in exposing defects, we compared SEGATE with some of the state-of-the-art test generation tools. We showed that SEGATE outperformed the other techniques on a dataset of 40 defects in library methods.

We plan to optimize the regex ($r_i$) generated by static analysis, in addition to improving the precision of the Static Analyzer to support complex numeric manipulation on strings. Other area of optimization is in reducing the test set by discarding highly similar strings. Determining the criteria of similarity would be of our interest [120]. Incorporation of user-defined specification constraints, when defining the criteria of similarity among inputs, could be one direction.

Further, a search-based approach to find behaviorally similar regex [30] may be useful to fully automate regex repair or recommendation. We would also be interested in integrating past techniques proposed on comment analysis [20, 154] with SEGATE to assess their effectiveness in inferring the input string's properties in deriving $r_s$.

# Chapter 6

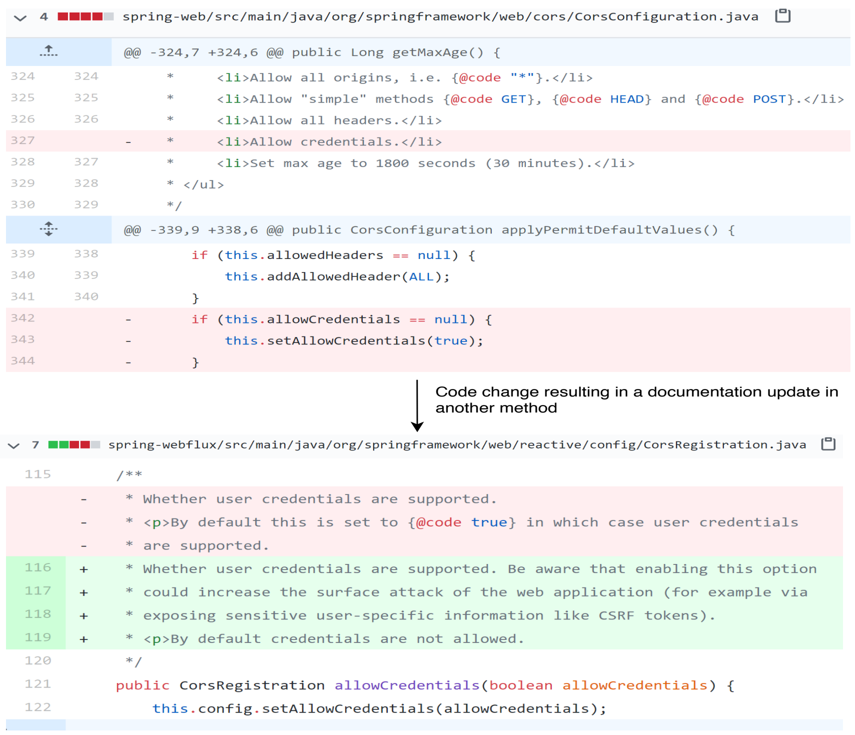
# Understanding Indirectly Dependent Documentation in the Context of Code Evolution

## 6.1 Introduction

A software system undergoes evolution over time. Changes to software in this evolution process may be attributed to factors such as enhancements, bug fixes, refactoring, and deprecation of entities. Modifications made at one point in the source code often require alterations at various other points to ensure the expected behavior. While developers use test-suites to verify that the changes to the code do not break any functionality, in many cases, less attention is given to making appropriate changes to the documentation associated with the functions. Such inconsistencies are undesirable for developers as well as users. A developer needs to obtain knowledge about code components for performing enhancements and other code-maintenance tasks. Documentation serves as a prominent source for acquiring this knowledge [39, 122]. Additionally, from a user's perspective, reading the documentation is preferable to deducing functionality from source code.

There have been research efforts to highlight inconsistencies between different code fragments or between code and its documentation (or comment) [62, 153, 115, 142]. Past work has focused on inconsistencies with function documentation when the directly associated function is modified. However, the documentation may be dependent on several other entities. Consider two snippets from a commit made in the Spring-Framework project, as shown in Fig. 15. The commit is sourced from an issue, SPR-16130 [136], which discusses altering the default access control to make a client request to a server. The field `allowCredentials` is by default set to enabled (or `true`), allowing any site to make an XHR request with credentials. The developers realize that this is not a secure configuration and resolve the issue by setting `allowCredentials` to disabled (or `false`) by default. Fig.15 demonstrates how this code change in one file induces a critical documentation update for the `allowCredentials` method in another file.

This work appeared at the IEEE/ACM International Conference on Software Engineering (ICSE), May 2021



```
4 spring-web/src/main/java/org/springframework/web/cors/CorsConfiguration.java
@@ -324,7 +324,6 @@ public Long getMaxAge() {
        *     <li>Allow all origins, i.e. {@code "*"}.</li>
        *     <li>Allow "simple" methods {@code GET}, {@code HEAD} and {@code POST}.</li>
        *     <li>Allow all headers.</li>
-       *     <li>Allow credentials.</li>
        *     <li>Set max age to 1800 seconds (30 minutes).</li>
        * </ul>
        */
@@ -339,9 +338,6 @@ public CorsConfiguration applyPermitDefaultValues() {
           if (this.allowedHeaders == null) {
               this.addAllowedHeader(ALL);
           }
-          if (this.allowCredentials == null) {
-              this.setAllowCredentials(true);
-          }
```

Code change resulting in a documentation update in another method

```
7 spring-webflux/src/main/java/org/springframework/web/reactive/config/CorsRegistration.java
     /**
-     * Whether user credentials are supported.
-     * <p>By default this is set to {@code true} in which case user credentials
-     * are supported.
+     * Whether user credentials are supported. Be aware that enabling this option
+     * could increase the surface attack of the web application (for example via
+     * exposing sensitive user-specific information like CSRF tokens).
+     * <p>By default credentials are not allowed.
      */
     public CorsRegistration allowCredentials(boolean allowCredentials) {
         this.config.setAllowCredentials(allowCredentials);
```

**Figure 15:** Snippets from a commit log showing a code change that triggers a documentation update in another file.

As a project evolves, there may be several changes such as the one discussed in this example. A developer may neglect to update the documentation, especially when it is not directly linked to the entity to which the changes have been made.

In this work, we study method documentation and commits logs of 11 open-source projects to observe trends on the prevalence of function documentations that are indirectly or implicitly dependent on entities other than the function with which they are associated. We find a substantial presence of such documentations. We also observe that developers are likely to leave the documentation inconsistent for a long period of time in the process of making updates to the project. Motivated by the extent and relevance of the problem of inconsistent documentation caused by indirect dependencies, we build a taxonomy of the types of documentation updates that indicate dependencies on other changes. We observe that the developers primarily enhance or fix the core description of the functionality. We present patterns that indicate potential relations between the source of an update, such as a code-change, and the affected documentation, in order to infer the cause of a dependency. We further discuss the implications of these patterns for developers from the perspective of building a warning system which indicates potential inconsistencies introduced on making updates.

To summarize, this work makes the following contributions:

- A study on the prevalence and the nature of indirect dependencies in function documentations.
- A comprehensive analysis of the nature of documentation updates over 1288 commit logs extracted from 11 open-source repositories. The analysis covers the role of developers in maintaining consistent documentation and builds a taxonomy of the nature of documentation updates.

```java
/**
   * Replaces the existing container under test with
   * a new container.
   * This is useful when a single test method needs
   * to create multiple containers while retaining
   * the ability to use {@link #expectContents()
   * expectContents()} and other related methods.
   *
   * @param newValue the new container instance
   * @return the new container instance
   * @throws IllegalArgumentException if the input
   * is not a valid encoded container
   * @since 1.0
 */
public Container resetContainer(Container newValue)
{    ..body }
```

**Listing 6.1:** Sample of a Javadoc comment for a method.

- Patterns that induce indirect dependencies in documentations. We discuss the implications of these patterns in building applications to reduce documentation inconsistencies in the context of code evolution.
- A publicly available annotated dataset of commit logs and methods with the above-mentioned analysis, to ease possible future extensions.

## 6.2 Experiment Design

The objective of this study is to analyze the nature of inconsistencies in documentations, as well as their causes, in the context of code evolution. We analyze method documentation and commit logs in open-source projects on GitHub that are implemented in Java. Java has been ranked as one of the five most popular languages on GitHub over the last five years [105]. We scope our study to analyzing documentation updates associated with methods, as method headers, in the code file. Listing 6.1 shows a Javadoc comment for a method, which is the typical documentation format for code files implemented in Java.

### 6.2.1 Research Questions

One expects a documentation to describe the entity with which it is directly associated. However, dependency of documentation on other entities would require maintaining the consistency of the documentation with other changes within or external to the project. We designed the first research question to study the prevalence of dependencies between a method's documentation and indirectly associated source code and other entities. These other entities include documentations associated with other methods, classes, interfaces, fields, or external sources, such as URL references to web pages. This brings us to the first RQ.

**RQ1:** *Are documentations written independently?*

**Table 14:** Dataset analyzed in the study.

| Repository | Star-Count* | # Commit logs extracted | # Commit logs analyzed |
|---|---|---|---|
| elasticsearch | 50.2k | 12742 | 344 |
| spring-boot | 49.5k | 6507 | 128 |
| RxJava | 43.2k | 376 | 64 |
| spring-framework | 38.5k | 3854 | 510 |
| guava | 38.3k | 545 | 72 |
| retrofit | 36.2k | 171 | 2 |
| dubbo | 33.1k | 1823 | 93 |
| MPAndroidChart | 31.3k | 70 | 2 |
| glide | 29.6k | 257 | 24 |
| netty | 24.3k | 1063 | 47 |
| gson | 18.3k | 57 | 2 |
| **Total** | | 27465 | 1288 |

* as of May 2020

We further study whether the maintenance of documentation over time matters to the developers and users of the library project. Hence, for the second RQ, we study discussions over issue reports and pull requests to observe whether documentation-related inconsistencies are highlighted.

**RQ2:** *How often is the dependence of documentation highlighted through user and developer discussions?*

Inconsistent documentations in a library can cause inconvenience to the developers and users of the library with respect to program comprehension. To address the third RQ, we study how often developers miss updating dependent documentations while making alterations to the repository.

**RQ3:** *How often is the consistency of dependent documentation with code maintained as the repository evolves?*

After studying the existence and importance of the problem of indirectly dependent documentation in the context of code evolution, we carry out a deeper qualitative analysis of the libraries through RQ 4 and RQ 5. This analysis reasons about the patterns under which a documentation update is needed and builds a taxonomy of the nature of documentation updates made as a project evolves.

**RQ4:** *What is the nature of updates made to documentation to maintain consistency?*

**RQ5:** *What factors influence a documentation to be updated with the evolution of repository?*

### 6.2.2 Data Collection

We selected 11 top-starred repositories of libraries from GitHub that are implemented in Java and documented in English. These repositories are listed in Table 14.

***Methods and Commits Extraction*** From the master branch of each repository, our script randomly chose source files and used JavaParser [135] to extract 50 methods and their Javadoc comments. We obtained 550 methods from 11 projects to study RQ1. To study the remaining RQs, over each of the 11 repositories, we used the Git API to extract commit logs from all branches spanning 2 years from June 15, 2018 to June 15, 2020. We extracted all the files changed in a commit. GitHub does not follow a mainline integration model; developers can create branches for various purposes and merge the changes into the stable branch once they are confident about it [18]. We only consider the non-merge commits to avoid analyzing potentially duplicate updates. This resulted in the extraction of 27,465 commit logs (Table 14). These commit logs were used to study the changes made to the repository and their effect on the documentations.

### 6.2.3 Analysis Procedure

We used a semi-automated approach to analyze the methods and the commit logs, where five of the authors performed the task of annotating the observations.

#### 6.2.3.1 RQ1: On the independence of documentation

We manually studied the 550 methods and their Javadoc comments extracted from 11 libraries to determine whether the documentation is only influenced by the associated method's source code or whether there exists another entity influencing the content of the documentation. The latter is a case of 'dependent' documentation. In order to characterize documentation dependencies, we conducted a preliminary study on a set of 100 randomly selected methods from the obtained dataset of methods. We checked whether each reference made by the target documentation could be mapped to the method source with which it is associated. In cases where we could not, we identified the content mapped to other entities, which we refer to as dependencies. In the process, we identified two key features that indicate a dependency. Accordingly, the annotator labelled a method documentation as 'dependent' if it satisfies at least one of the following criteria.

- The documentation has references to other entities that are object types, methods, fields, file paths, or URLs.
- The description of the documentation explains more than what the associated method directly implements. This description is either sourced from the invoked methods or other external factors.

#### 6.2.3.2 RQ2-5: On the importance of maintaining consistency and understanding the nature of and the reasons behind inconsistencies in the context of implicit dependencies

As the focus of this study is the documentation associated with a method, we shortlisted commit logs from the 27,465 extracted logs which were more likely to highlight the indirect dependencies of the documentation. Consider a scenario where both a method's source code and its documentation are updated; it is likely that the documentation update is induced by the change made to the associated method. However, in cases where only the documentation of a method is being updated, there are high chances of it being induced by changes made elsewhere. Further, these changes may be sourced from either the same commit in which the documentation is updated or from one of the previous commits. Based on this idea, we shortlisted commits that contained at least one method for which only the documentation had been updated, that is, without any change made to the corresponding method's implementation (filtration details in Section 6.2.3.2). The annotators manually analyzed these commits, as described later in this section.

***Commit Filtration*** To filter out the unwanted commit logs, we followed a two-stage process. For a given list of commit IDs and the corresponding list of altered files (described in Section 6.2.2), the first stage uses the Git API to get the parent commit ID of each commit. We used the `git diff` command to compute the lines changed in a file between the two commits, `commitid1` and `commitid2`. These changes are in the form of insertion and deletion of lines. A script stores all the changed lines in JSON format, with keys as `commitid1_commitid2_filename` and the values being the arrays of line numbers of insertions and deletions.

To obtain methods that have been altered in a commit, the second stage uses JavaParser alongside Git API to process the JSON file. From each key, our script parses the two commit IDs and filename. It then extracts the two versions of the file corresponding to the two commits. On each file version, we used JavaParser to visit all top-level class's methods and get the start and end lines of the code and its documentation. A script compares these lines with the lines inserted or deleted to extract the list of methods changed between two commits for a given file. The script shortlists commits that contain at least one method in which only the documentation was changed. Note that there may still exist other methods having code-related changes in such commits. Applying this filtration process, we shortlisted 1,288 commits (Table 14).

***Features Captured*** We perform a qualitative analysis on the shortlisted commits to capture various features associated with a commit log. As the study focuses on analyzing dependencies in documentation associated with methods, we use the following related terms throughout the chapter.

*A **source** is an entity, such as changed code or a referred URL, which influences the documentation of an indirectly related method.*

*A **target** is a method whose documentation has been affected by an indirectly related entity (the source).*

Each annotator was provided with a JSON file that contained shortlisted commit IDs and corresponding lists of methods for which only documentation was modified. These methods were used to identify the sources and targets. Since we want to capture the maintenance of documentation consistency in the context of project evolution, the annotators analyzed only the differential part of the documentations in the given commit to log the observations. A documentation was labelled as 'dependent' as per the two criteria discussed in Section 6.2.3.1. Additionally, if the commit difference suggests that the documentation change occurred due to a change made elsewhere, such documentation was also labelled as 'dependent'. The method associated with this documentation was labelled as a target. Note that not all shortlisted commit logs necessarily indicate an indirect documentation dependency. Although they may appear in our shortlist, documentation updates such as grammatical fixes or formatting changes are not dependencies. Further, documentation updates may be directly sourced from previous changes made to the associated method. As we scope our study to indirectly induced documentation inconsistencies, we ignored such direct updates during the analysis.

*RQ2: On documentation being discussed:* We studied whether the maintenance of documentation matters to the developers and the users of the libraries. The annotators captured whether the commit log under analysis is linked to any pull request or issue report such that the documentation is the main topic of discussion over the thread.

*RQ3: On maintaining the documentation consistency as the repository evolves:* We studied the prevalence of cases in which a change is made to the project while the indirectly dependent documentations are overlooked and changed much later. Hence, an annotator captured whether a documentation update is sourced from a change made in the same commit or in a previous commit. The commit-change description and the discussion thread on the related pull request or issue report were used to obtain hints on the cause of a documentation update. This aligns well with the most frequently highlighted theme of 'change description' by developers, as listed by Ram et al. in their study on what constitutes a good code change from the perspective of change-reviewability and change-comprehension [112]. If the annotator did not find the source of documentation update in the same commit, he/she searched the linked discussion thread for potential references to previous commits and studied the commit history of the method to find the source of update. Further, on obtaining the previous commit, the annotator also noted the time between the introduction and the fixing of the documentation's inconsistency.

*RQ4: Nature of documentation updates to maintain consistency:* To understand how documentation is influenced by various indirect changes, we studied the nature of updates made to the documentations. As observed in our dataset, documentation (a Javadoc comment) typically consists of a description of the method's functionality, parameters, returned output, and extra information for detailed understanding. We identified 6 main categories of the nature of documentation updates described in Table 15. We further identified 8 sub-categories for 'description update' (Table 15).

All the categories have been defined to be exclusive of each other, as also validated over our dataset used for the analysis. This implies that every single update made to a documentation gets categorized uniquely. However, it may still be possible for different parts of a documentation to have changes of different sub-nature, in the 'description update' category. A typical commit log targets a single objective. Hence, every dependent documentation update was assigned with a single main category. Often, one type of documentation update triggered by a code change can result in other types of updates elsewhere. For instance, consider a case where the introduction of a new method leads to the deprecation of an older method. As a result, the developer adds a deprecation note to the older method. This leads to the replacement of references made by the documentation of other methods to the older method (now deprecated) with the new method. Hence, one would observe two categories of documentation updates, 'mark deprecated' and 'description update'. However, we know that the root type of the documentation update in this case is 'mark deprecated', despite the presence of other updates. Therefore, for clarity, we captured only the root update category for a commit log.

In case of a 'description update', it may be possible to assign multiple sub-categories to the documentation, owing to updates in multiple sections. For instance, if an update is made to the description of the functionality, and additionally, extra references are introduced for further clarification, then both sub-categories, 'behavioral description' and 'extra suggestions description', shall be noted. An annotator assigned appropriate category and sub-categories to the updates in the target's documentation in the commit.

*RQ5: Relations that trigger documentation updates:* To understand the scenarios for which documentation is more likely to be updated, we studied the relations between the source and the target. We obtained patterns that can be harnessed to build applications. The annotators searched for the following relations between the source and the target and logged patterns of these relations for each commit log.

*i. Referential relation*: We captured whether the source or the target makes direct references to each other in their respective documentations. This relation indicates that if either of these entities references the other, then a change in one will potentially require updating the other's documentation. Hence, a significant prevalence of such a pattern can be useful in automating the process of shortlisting targets, given a source. While analyzing a commit, for the extracted source and target pairs, the annotator logged one of the following

**Table 15:** Nature of documentation updates.

| | Category Description | Example of a documentation update |
|---|---|---|
| 1. | **Refactoring**: Structural changes made to the references in the documentation, which do not change the description of the functions or semantics. | Elasticsearch commit 1a5e72e [45]: *"…When created through #**withLocalReduction(SearchRequest, String[], String, long, boolean)**, this method returns.."* replaced by *"…When created through #**crossClusterSearch(SearchRequest, String[], String, long, boolean)**, this method returns.."* due to the renaming of the referenced method. |
| 2. | **Description update**: Semantic update to the core description of the method. | |
| | 2a. **Behavioral description**: Updates in the functionality's intent description without making any direct references to entities. This may occur due to change in the source's behavior without directly referencing it. | MPAndroidChart commit fcc5af7 [98]: *"Callbacks when the chart is scaled / zoomed via pinch zoom gesture."* replaced by *"Callbacks when the chart is scaled / zoomed via pinch zoom / **double-tap gesture**."* as the support for double-tap zoom is added in another class that invokes the target method. |
| | 2b. **Referential description**: Updates in the references to entities for enhancing the functionality's intent description. These references may also be to class comment to get further insights. | Glide commit ed20643 [50]: *"..Previous calls to #apply(**RequestOptions**).."* replaced by *"..Previous calls to #apply(**BaseRequestOptions**).."* where BaseRequestOptions class is added as the parent of RequestOptions. |
| | 2c. **Extra suggestions description**: References to other entities for additional details. | Spring-framework commit 859923b [137]: Adding *"**@see #getBeanType()**"* to the documentation of *getBeanName()*, as extra information. |
| | 2d. **Parameter description**: Updates in the description of method parameters and constraints induced on them by other entities. | Elasticsearch commit 3f2a241 [44]: Addition of *"@param requestId see ResponseHandlers#add(ResponseContext) for details"* to the parameter description, on addition of ResponseHandlers class. |
| | 2e. **Return value description**: Updates in the description of the return-values and their constraints. | Guava commit 21cfbea [6]: Clarification added on the return type by referencing SortedMap in the documentation *"the returned map itself is not necessarily a SortedMap"*. |
| | 2f. **Example description**: Updating examples to describe the method usage or functionality, which may also involve configuring other entities. | Guava commit 3dfee64 [58] : *"Example: Files.fileTraverser(). **breadthFirst("/")** may return files … ["/", "/etc", …"* replaced by *"Example: Files.fileTraverser().**depthFirstPreOrder(new File("/"))** may return files … ["/", "/etc", "/etc/config.txt", …"* to include a more common example. |
| | 2g. **Alternative reference description**: Updating the documentation with a reference to another method that can serve as an alternate to the target method under certain specific requirements. | RxJava commit 6ba932c [117]: Documentation of target updated with an alternative reference- *"Similar to **Objects.requireNonNull** but composes the error message.."*. |
| | 2h. **TODO notes description**: Updating a note left in the documentation for future action having direct/indirect references to other entities. | Guava commit af3ee1c [59]: Documentation is added with a future action due to a code-change made elsewhere- *"TODO(user): remove the addNode() calls, that's now contractually guaranteed"*. |
| 3. | **Mark deprecated**: Deprecation of the target method, leading to referencing an alternative in the documentation. | Spring-boot commit a63ab46 [138]: Deprecation note added to an existing method when a new method is introduced- *"@deprecated in favor of #setRSocketServerCustomizers(Collection) as of 2.2.7"* |
| 4. | **Exception type addition/update**: Declaring or updating exception types in the documentation that may get thrown by the target method. | Spring-framework commit 71f3498 [7]: Exception was declared in the Javadoc of a method as one of the callee method's code was updated- *"@throws DecodingException if the metadata cannot be decoded"*. |
| 5. | **Referential typo fix**: Typo fixes in the documentation made in references to the source entities. | Netty commit 8f01259 [100]: Incorrect reference to a method in the documentation fixed from *set**max**HeaderListSize(long)* to *set**Max**HeaderListSize(long)*. |
| 6. | **Version update**: Updating the documentation in the context of certain platform or version dependencies. | Guava commit 7fdf1a6 [60]: The documentation is updated with a special instruction for Java 9 users- *"Java 9 users: use java.util.Objects.requireNonNullElse(first, second) instead"*. |

**Table 16:** Free Marginal Kappa scores to evaluate the inter-rater agreement.

| Feature Analyzed | Kappa Score |
|---|---|
| Documentation has indirect dependency | 0.88 |
| Documentation update triggered by change made in the same or previous commit | 1 |
| If the update is sourced from a discussion on the documentation | 0.89 |
| Update induced from the change made in same or different file | 0.76 |
| Nature of documentation update | 0.84 |
| Call-graph relation-type | 0.74 |
| Reference relation-type in the documentation | 0.85 |
| Inheritance relation-type | 0.93 |
| Interface relation-type | 0.93 |

observations: a) target's documentation references the source, b) source's documentation references the target, c) both, d) none.

*ii. Call-graph relation*: A call-graph captures the control flow relation between the methods calls made in a program. We captured this relation between the source and the target based on the idea that such relations would be reflected in the program's behavior, and hence are likely to be documented by the developer. Therefore, in the context of alterations to the code, the documentation may also require updates. The annotator logged one of the following observations per commit: a) target in the call-graph of source, b) source in the call-graph of target, c) none. These relations were captured on the call-graph generated using the *Soot Framework* [76].

*iii. Inheritance relation*: We investigated whether it is common for the respective classes of the source and the target to share a relationship based on inheritance. Certain properties may get propagated by this relation, thereby creating a dependency between the source and the target. Hence, for each commit, the annotator captured the following observation on the classes to which the source and the target belong: a) target's class inherits from source's class, b) source's class inherits from target's class, c) classes of source and target share a common parent, d) source and target belong to same class, e) none.

*iv. Interface relation*: As interfaces provide abstractions that the implementing classes must adhere to, we studied the prevalence of such a relation between the source and the target's class or interface in an attempt to infer the cause of the dependency. For every commit, the annotator noted the corresponding class or interface of the target and source and logged one of the following observations: a) target's class implements from source's interface, b) source's class implements from target's interface, c) classes/interfaces of source and target implement/extend a common interface, d) source and target belong to the same interface, e) none.

***Inter-rater agreement*** An agreement mechanism was followed batch-wise in the process of analyzing the data. Initially, each of the five authors annotated 10 commit logs. Then a group discussion was held among the authors to discuss all 50 logs, to come to an agreement and check consistency in the understanding. A second batch of 10 logs, analyzed by each annotator, was validated by another annotator. To capture the inter-rater agreement among the annotators, 10 randomly picked commit logs from the dataset were provided to be independently analyzed by each of them. We computed the free-marginal kappa agreement scores (Table 16) [150] on these annotations [114]. A kappa score of 0.88, which indicates "almost perfect agreement" [150], was obtained for the classification of documentation updates as indirectly dependent or independent. On obtaining a thorough understanding through this exercise, all annotators continued to annotate a subset of commit logs and marked their confidence levels as high, medium, or low, to associate with each annotation.

```java
/** Returns the hex string of the given byte array
    representing a SHA256 hash. */
public String sha256BytesToHex(byte[] bytes) {
    synchronized (SHA_256_CHARS) {
      return bytesToHex(bytes, SHA_256_CHARS);
    }
}
```

**Listing 6.2:** Javadoc comment that is dependent on the invoked method without directly referencing it.

All logs marked as 'low' were resolved in a group discussion and the logs marked as 'medium' were validated by another annotator.

## 6.3 Findings

With the design and experiments to explore the mentioned RQs, we discuss the findings in this section. We have made all the findings of the study, along with the dataset and the scripts used to prepare it, available[1].

### 6.3.1 RQ1: Are documentations written independently?

For 341 of the 550 analyzed Javadoc comments of methods, we observed the prevalence of dependencies on indirectly associated source code and other entities. Considering the two criteria mentioned in Section 6.2.3.1 to mark a documentation as dependent, for 321 cases, the Javadoc comment had references to the source of dependencies, while for 20 cases the dependency was purely descriptive, without references to the source. Fig. 16 shows a project-wise distribution of dependencies with and without references. Across all projects, the occurrence of dependent Javadoc comments ranged from 36% to 74% (overall 62%). These numbers indicate a significant presence of dependencies that require the attention of the developers in the context of making updates to the repositories.

Listing 6.2 shows a method for which the dependent Javadoc comment describes the method that has been invoked by `sha256BytesToHex` without directly referencing it. Fig.16 represents the share of such cases in red. Cases like these make it difficult to automate the identification of sources and potentially require support from natural language processing techniques, in addition to program analysis. The documentation containing references had the following types of references: 1) method, 2) field, 3) class object and exception types, 4) enum types, 5) URL to external resources, and 6) package path. Fig. 17 gives a taxonomy of the nature of dependent Javadoc comments observed while studying for RQ1. Most of these categories are similar to those stated in Table 15; however, note that the latter describes the nature of 'documentation updates', discussed later, while the former describes the nature of dependent components in the whole 'documentation'.

**Finding 1:** *There is a substantial presence of indirect dependencies in the documentations, which requires the attention of the developers in the context of making updates to the repositories.*

[1]`https://github.com/pag-iiitd/DocDependency`

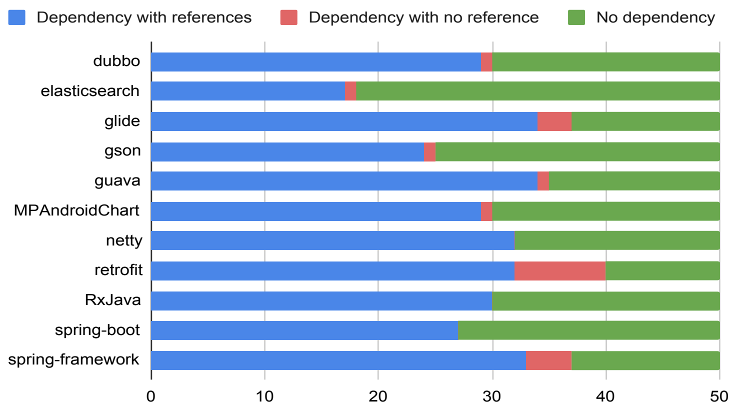
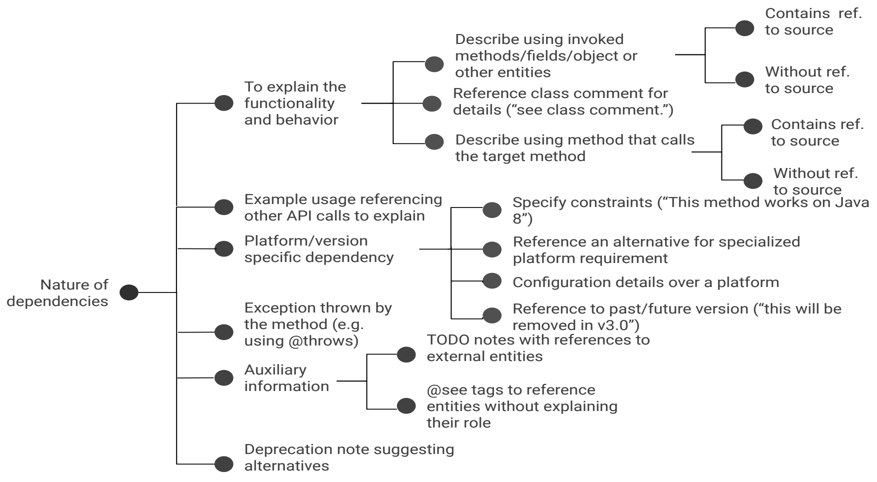


**Figure 16:** Project-wise distribution of indirectly dependent and independent method Javadocs studied for RQ1 with 50 methods analysed from each of the 11 projects.



Nature of dependencies
To explain the functionality and behavior
Describe using invoked methods/fields/object or other entities
Contains ref. to source
Without ref. to source
Reference class comment for details ("see class comment.")
Describe using method that calls the target method
Contains ref. to source
Without ref. to source
Example usage referencing other API calls to explain
Platform/version specific dependency
Specify constraints ("This method works on Java 8")
Reference an alternative for specialized platform requirement
Configuration details over a platform
Reference to past/future version ("this will be removed in v3.0")
Exception thrown by the method (e.g. using @throws)
Auxiliary information
TODO notes with references to external entities
@see tags to reference entities without explaining their role
Deprecation note suggesting alternatives

**Figure 17:** Nature of dependent updates observed for RQ1.

### 6.3.2 RQ2: How often is the dependence of documentation highlighted through user and developer discussions?

Of the 1,288 shortlisted commit logs, we observed that for 592 cases, the commit involved the updating of at least one method's documentation (the target) due to modifications made at other places (the sources). A single commit may have multiple targets and sources arising from cases where a) a set of modifications collectively influence documentation of the target, which leads to multiple sources, or b) a single change induces documentation updates for multiple methods, resulting in multiple targets. The nature of documentation update was found to be highly similar in the case of multiple targets.

**Finding 2:** *46% of the commit logs in the dataset resolved indirect dependencies in the method's documentations.*

This observation indicates that the maintenance of documentation matters to the developers. We further observed that 189 of 592 (32%) commits were sourced from discussions, available on the thread of issue reports or pull requests, that were specifically on documentation. The following are two such comments quoted from the discussion threads:

*'The link to the HLRC documentation for the async Put Lifecycle method was never updated to the correct link. This commit fixes that link.'*

*'Questions on how to work with ActionPlugin#getRestHandlerWrapper() come up in discussion forums all the time. This change adds an example to the Javadoc how this method should/could be used.'*

**Finding 3:** *Maintenance of the documentation matters to the developers and users of the repositories, as evident from the discussion threads.*

### 6.3.3 RQ3: How often is the consistency of dependent documentation with code maintained as the repository evolves?

Ideally, any change made to the repository should be reflected in the dependent documentations at the same time. However, we observed that in reality, the documentation at a point is often left inconsistent with the committed code. Of the 592 commit logs containing dependent documentation updates, in 77 cases (13%), the documentation update was sourced from changes made in past commits. Further, we observed that this dependent documentation may be left inconsistent for anywhere from 0 to 1988 days since the change that induced the documentation update, with a mean resolution duration of 469.8 days and standard deviation of 565.7 days.

**Finding 4:** *As the repository evolves, dependent documentations may be left inconsistent for a long time.*

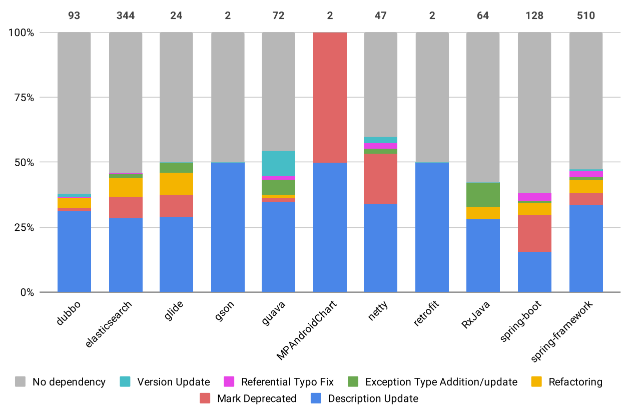
### 6.3.4 RQ4: What is the nature of updates made to documentation to maintain consistency?

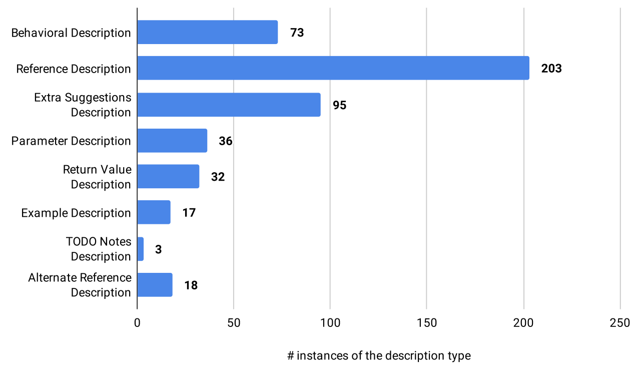
For 325 of the 592 commit logs (54.9%) containing dependent documentation updates, the target existed in a different file from the source. 48 of the 325 cases had multiple targets of which some belonged to the same file as the source and others belonged to different files. The sources may also be external entities existing in a library or a web resource different from the target, as observed for 81 of 592 cases (13.7%). Such external dependencies can pose challenges in tracking the updates. One such documentation update containing a link to a web-page is quoted as *'This can leave your server implementation vulnerable to "https://cwe.mitre.org/data/definitions/113.html" CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting').'*

**Finding 5:** *Documentation may contain updates from external sources (not present in the target repository).*

Fig. 18 shows the observed distribution of the nature of updates made to method documentation. Fig. 18(a) indicates the project-wise existence of different update categories described in Table 15. The grey fraction of each bar indicates the percentage of shortlisted commit logs with no indirectly dependent documentation updates; these documentation updates were either directly influenced by the associated method or grammar or formatting fixes. Among the dependent documentation updates, 'description update' dominated across all projects, followed by 'mark deprecated'.

Each commit log with a 'description update' (386 of 592 commits) may be further described by one or more sub-categories. We analyzed the description updates (Fig. 18(b)) and observed that in most cases, the





**(a)** Project-wise distribution of nature of documentation updates. **(b)** Distribution of sub-categories of 'description update'.

**Figure 18:** Nature of documentation update.

update enhanced or fixed the description of the functionality by using references, covered by the category 'referential description' (203 of 386 cases). The second most popular description update type was 'extra suggestions description' (95 of 386 cases), to add or update supplementary details by describing or referencing entities for added clarity about the functionality. This was followed by 'behavioral updates' to clarify or fix the functionality description. These observations indicate that while in most cases the sources influence updates of the description of the functionality, the developers also seek to maintain additional details that normally act as supplementary references.

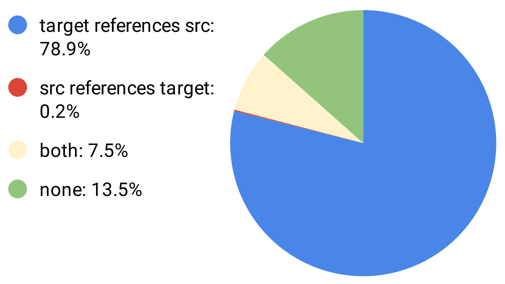
**Finding 6:** *Developers seek to update the documentation in a variety of ways. They mainly enhance or fix the core description of the functionality, indicate deprecation by referring to an alternative, or update the supplementary details for better clarity about the method.*

### 6.3.5 RQ5: What factors influence a documentation to be updated with the evolution of repository?

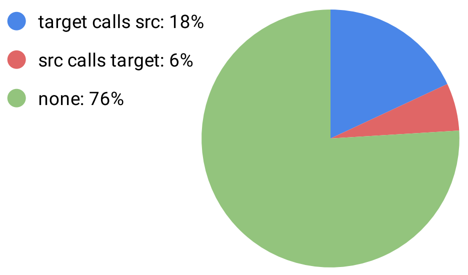
To reason about the dependency caused in documentations, we analyzed patterns in four types of relations between the source and the target methods and their classes/interfaces. Fig. 19 shows a distribution of source and the target associations across the four relations, as observed in our dataset.

Referential relation captures the association of source and target through a reference to one in the other's documentation. Fig. 19(a) indicates that for 86.6% of the commits, either the target's or the source's documentation mentioned the other. In most cases, the target referenced the source (78.8% cases). Hence, an alteration to the source, such as renaming the source method, may induce an update to the target's documentation. The presence of such references may prove to be useful in identifying the source given a target or a target given a source. The latter is especially useful when the developer needs to identify points (the targets) where documentation (or code) needs to be updated after a source is modified.

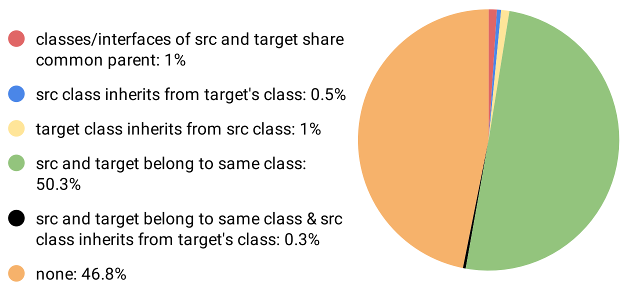
**Finding 7:** *The documentation of the source method often has references to entities influencing it.*



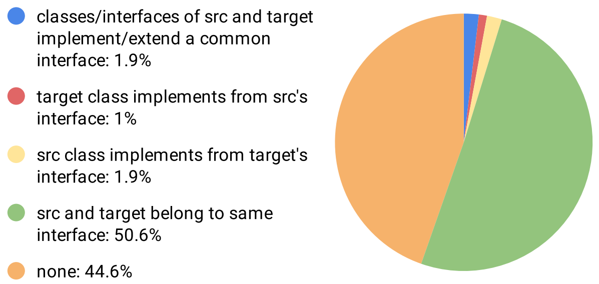
**(a)** Referential relation



**(b)** Call-graph relation



**(c)** Inheritance relation



**(d)** Interface relation

**Figure 19:** Relation between the source and the target as observed in 592 commit logs.

To further analyze the cause of such references, we explored three types of programmatic relations between source and target. Call-graph relation captures if the two are related by a caller-callee relation (which may occur at any depth in the call-graph). We observed that for 24% cases, such a relation existed (Fig. 19(b)), with the target calling the source in most of these cases. This indicates that the documentation of the target often describes or derives information from the called entities, which induces a dependency.

We further analyzed if classes or interfaces corresponding to source and target are related. A dominant case was observed in which the source and the target are present in the same class/interface (Fig. 19(c),(d)). This can be explained by the usual practice of entities, such as methods or class fields, within a class interacting with each other. As a result, their documentations are likely to have details that are dependent on one another. There were a few instances where the corresponding classes/interfaces of the source and target extended/implemented the same parent class/interface, shared a parent-child relation, or implemented the other's interface. Inheritance aims at propagating properties that get reflected in a child class. A change in either the parent or its children must not violate these properties. Similarly, an interface provides abstractions that the implementing classes must adhere to. These relations explain possible causes of dependencies in documentation, and may also be useful in mitigating inconsistent documentation, as we discuss in Section 6.4.

**Finding 8:** *Several programmatic relations such as caller-callee pattern, inheritance of properties, and implementation of abstractions propagated by interface may influence the content of the documentation of an entity.*

Of the 592 commits having dependent documentation updates, we observed that at least one of these four relations categorized to a value other than 'none' for all but 11 cases. Hence, searching for such patterns may help in identifying sources and targets. For the 11 commits where none of the patterns existed, we analyzed how else the source and the target were related. We observed the following patterns.

- *Non-programmatic source which isn't referenced in the target's documentation.* Example: Example code in the documentation was changed based on the recommended convention in an RFC documentation.
- *Source and target accessing a common entity.* For instance, consider a class field which is accessed for manipulation by the source and target methods, with the documentation being derived from this behavior.

We plan to explore these patterns on the entire dataset as future work in order to draw further insights.

## 6.4 Discussion

The discussed observations may find a direct use case for developers in the form of a recommendation system to suggest methods (targets) for which code or documentation should be altered when a developer makes code changes to other entities (sources). This would help prevent documentation inconsistencies that arise as a project evolves. Some patterns that can potentially be leveraged for the application are discussed in this section.

In 6% of cases with dependent documentation updates, we observed sources calling targets. This pattern is directly useful for shortlisting methods to recommend for a documentation update using the program's call-graph. On the other hand, 18% of cases with documentation updates involve targets calling sources. This pattern could be applied to shortlist the targets for a given source by maintaining a linkage graph for the repository where edges exist from a programmatic entity to all other entities that call or use it; the graph may be dynamically updated for every code change. Hence, as a code change occurs, all nodes directly or indirectly linked to it can be shortlisted to check for documentation inconsistencies. A similar linkage graph can be maintained to reflect inheritance or interface relations. For a scalable and a meaningful recommendation, the described approach can be combined with heuristics based on referential relations, which were found to exist in abundance. For instance, the shortlisted linked entities could be further processed such that the methods referencing the source in their documentation or whose references appear in the source's documentation can be shortlisted for updating the documentation. Several such heuristics can be developed to leverage the observed patterns.

The recommendation may be extended to suggest documentation changes for the identified targets. This would require studying relations between the nature of code or documentation change made at the source and the nature of the induced documentation update observed at the target. Combining these relational insights with natural language processing techniques could aid in building an IDE plugin for recommending documentation updates in addition to identifying methods for which a documentation update is required.

## 6.5 Threats to Validity

**Construct validity** Inferences of the source and the nature of documentation update are purely on openly available artefacts such as discussion threads and commit history. Other missing factors, such as in-person discussions between developers, may have influenced code and documentation changes.

The selection of only library projects was incidental. Applying our criterion to pick top-starred projects resulted in only libraries and frameworks. One explanation for this trend could be that libraries and frameworks are meant to be reusable for further development, and hence, more developers use or star them. Extending our analysis to non-library projects would make for interesting future work.

**Internal validity** To mitigate the possible subjectivity in the analysis, which has been done by 5 annotators, we followed a phase-wise approach to build common understanding. We evaluated this understanding through an inter-rater agreement exercise (Table 16). Further, logs marked with medium and low confidence by an annotator were given to other annotators to analyze independently (Section 6.2.3.2).

For four cases with untraceable sources, we take a conservative approach by not considering the documentations to have indirect dependencies.

**External validity** The filtration criteria used to extract the commit logs (Section 6.2.3.2) may not cover all cases of documentation updates that have been indirectly influenced in a project. However, the criteria only give an underestimate; the actual number of cases of dependent documentations is likely to be higher, which makes the relevance of the problem even stronger.

The study has been conducted on projects implemented in Java. As Java is among the top 5 languages on GitHub, our findings may nonetheless be useful to a significant fraction of the developer community. With a considerable amount of manual analysis involved and the underlying language-dependency of the APIs used for data processing, we limited the study to 11 projects. With the discussed patterns found by the study, it opens an opportunity to automate some components of the analysis and explore projects from more languages and diverse classes.

## 6.6 Conclusion

We conducted a comprehensive study on 11 open-source projects to analyze the prevalence and nature of documentation dependencies on entities other than the associated function. We analyzed 550 Javadoc comments and observed over 62% of them to be indirectly dependent. We further analyzed 1288 commit logs to categorize the nature of updates made to documentations, as induced by code-evolution changes and external changes. To reason about the factors that induce these inconsistencies in the documentation, we studied four types of relations between the source and the target. We observed the presence of references in the documentation to be a dominant reason for the existence of dependencies. Analyzing the programmatic relations between the source and the target, we suggested applications of these patterns. One such application is a recommendation system to suggest methods for which documentation should be updated when a developer makes changes in other sections of the repository.

The findings of the study open several interesting problems. One such problem would be to track updates in external sources that can trigger documentation updates in the repository. As the domain of external sources is huge and mostly non-programmatic in nature (such as web-pages), taking note of their updates and influence on the target would be a challenge. Further, there may exist unexplored patterns relating the source and the target that can be leveraged to mitigate the problem of inconsistent documentation. Combining programmatic features, extracted by program analysis, and machine learning approaches may aid in learning these patterns.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

The thesis statement was motivated by the problem that the inconsistencies may arise from either the implementation or the documentation of a functionality and hence, assuming one as a baseline to fix another would be unreliable. Furthermore, formulating complete and precise specifications is a hard problem. To address these challenges, we have proposed to explore external resources, other than the specification and the implementation, to highlight the inconsistencies. In this direction, we have proposed test generation techniques that make use of resources such as test suites associated with similar functionalities, external documentations such as RFCs and the domain knowledge of developers. Our proposed tool, METALLICUS, was shown to reveal 46 defects in open source libraries, by recommending test cases obtained by mining similar functions across libraries. Another tool, SEGATE, revealed 80% of the defects in the evaluation dataset and additional 6 new defects, by leveraging the developer's domain knowledge and external resources to model structured inputs to highlight their inconsistencies with the input constraints inferred from the implementation. The number of defects revealed in real programs by the tests generated by these techniques indicates the effectiveness of leveraging external resources. These defects ranged in nature from highlighting behavioral defects in the functionality to ambiguities in the documentation.

We also proposed to understand the nature and the cause of introduction of the inconsistencies between an implementation and dependent documentation in the process of project evolution. To do so we leveraged the commit-logs and the developer discussions on code-projects to build a taxonomy of the nature of documentation updates. We studied patterns to potentially explain the dependencies. We observed the presence of references in the documentation to be a dominant reason for the existence of dependencies. Analyzing the programmatic relations between the source and the target, we propose that patterns can be leveraged to build applications that reduce the introduction of such inconsistencies.

The proposed techniques and findings indicate that external resources and meta-data can be leveraged to highlight the gap between a program's implementation and its documentation and understand how these gaps are introduced.

### 7.1.1 Discussion

The ideas proposed in this dissertation are dependent on the existence of resources, such as test suites and commit logs, which are usually found in abundance. For instance, the effectiveness of Metallicus depends on the existence of similar methods and their corresponding test suites. As a consequence, the mining approach would be better suited to test utility libraries, that contain commonly used functionalities, over applications that contain unique or less common functionalities, for which a similar function may not exist. Furthermore, Metallicus assumes the returned matches to be functionally similar. It would be an interesting direction to explore how semantically different functions with syntactical similarities could be tested using this mining approach.

The idea incorporated in the tool, Segate, validates the feasibility of capturing the gap between an implementation and its specification to generate test cases. Segate has been scoped to test the methods taking structured strings as inputs. The technique has been specific to modelling string structures and the data flow analysis has been designed to capture the constraints in the form of regular expressions. Segate has been built on the fundamental idea of analysing the implementation and the documentation to build two representative models to capture the implicit and explicit constraints on the inputs. The difference between these models is leveraged to generate test cases to highlight the gap. It would be an interesting line of research to adapt the fundamental approach to other data types. One challenge would be to identify a suitable representation for a data type other than string. For instance, numeric inputs may be represented by linear or non-linear constraints. This would require a different data flow analysis to capture the relevant constraints. In fact, identifying a generalised representation, independent of the underlying data type would allow building the tool for a wider utilization of the proposed idea. Another interesting line of research would be to understand how inputs of different data types or attributes of user-defined data types interact in an implementation and how those interactions can be captured in a constraint model. There may be several other interesting extensions to the proposed techniques, some of which we discuss in the next section.

## 7.2 Future Work

### 7.2.1 Compatible test translation across languages

The test recommendation approach presented in Chapter 4 has scope for improvement with respect to mapping data types across libraries and languages. The current state of the proposed tool, Metallicus, expects a one-time preparation of type hierarchy per language and a mapping of equivalent primitive data types across languages. This is an essential requirement of Metallicus to map the parameters across similar functions. It would be useful to have an automated approach to perform this mapping so that the tool could be more conveniently extended to more languages. Past patterns from code migration across languages or mining information from online forums and question-answer sites such as Stack Overflow may be leveraged to fetch the information for data type mappings.

Furthermore, even within a language the task of test-adaptation across functionalities can bring in several challenges pertaining to the data types. For instance, consider a function that returns a String, while its similar function returns an array of characters. Essentially, both the outputs are representations of sequence of characters. Consider another pair of similar functionalities where one function returns a Hashmap while another returns a Json string. Both the outputs are used to represent key-value pairs. It would be appropriate to have an approach that can extract logical similarity patterns between the data types in a given context of the functionalities. These patterns would be useful to automatically prepare templates to adapt test cases across functions.

### 7.2.2 Approaches to reduce inconsistencies

Having proposed some patterns of dependencies to indicate the potential causes of the introduction of inconsistencies in the documentation (Chapter 6), the next line of extension would be two-phased. Firstly, the approach requires to track updates in external sources that can trigger documentation updates in the repository. A recommendation tool can use the patterns found in the study and build heuristics on top of those to identify highly probable functions that are impacted by a change made in other functions. Further, there may exist unexplored patterns relating the source and the target that can be leveraged to mitigate the problem of inconsistent documentation. Combining programmatic features, extracted by program analysis, and machine learning approaches may aid in learning these patterns. Secondly, learning the patterns of changes to the program and the corresponding change in the documentation may further be useful in recommending what update should be made to the documentation to avoid the introduction of inconsistency.

### 7.2.3 Leveraging resources to test machine learnt software models

This dissertation has focused on leveraging resources to test traditional software. It would be interesting to extend the proposed ideas to test machine learnt models. For instance, the approach of leveraging similarities in the functionalities may be extended to leveraging similarities in different models that are meant to perform the same task. For instance, consider a decision tree classifier and a perceptron model that classify an image as a human or a non-human. These similarities can be leveraged to perform a differential testing on different models trained on same data to infer the gaps and potentially highlight inaccuracies and other weaknesses in these models. This form of testing may be especially useful when a formal specification and gold standard data may not be available, which are common scenarios in the machine learning domain.

Segate (Chapter 5) performs data flow analysis on the implementation to extract information and takes a differential of it with the specifications for smarter test generation. It would be interesting to see how such a differential approach may be adapted to machine learnt models. For an analysis to extract meaningful information, a model needs to be interpreted or converted to an interpretable surrogate model. The information may then be used to take a differential with the specifications. This would allow the generation of test cases to highlight the weaknesses in the model or the specification. Another challenge here would be to propose an appropriate representation of the information extracted from the model and the specification in order to be able to take a differential; this logic is analogous to the regular expression as a representation that is used by Segate.

# Bibliography

[1] `http://justusrandolph.net/kappa/`.

[2] Awesome java. `https://github.com/akullpp/awesome-java`.

[3] Awesome python. `https://github.com/vinta/awesome-python`.

[4] Github 2017 statistics. `https://octoverse.github.com/`.

[5] Pygithub library. `http://pygithub.readthedocs.io/en/stable/index.html`.

[6] *Guava. Commit 21cfbea*, 2019 (July 16, 2019).

[7] *Spring-framework. Commit 71f3498*, 2019 (September 2, 2019).

[8] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *International Conference on Computer Aided Verification*, pages 150–166. Springer, 2014.

[9] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[10] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 225–236. ACM, 2014.

[11] Muath Alkhalaf, Tevfik Bultan, and Jose L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 947–957. IEEE, 2012.

[12] Suriya Priya R. Asaithambi and Stan Jarzabek. Towards test case reuse: A study of redundancies in android platform test libraries. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, pages 49–64, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] Abdulbaki Aydin, Muath Alkhalaf, and Tevfik Bultan. Automated test generation from vulnerability signatures. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 193–202, 2014.

[14] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *International Conference on Computer Aided Verification*, pages 255–272. Springer, 2015.

[15] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 157–166, New York, NY, USA, 2010. ACM.

[16] 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*, 41(5):507–525, May 2015.

[17] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the ide: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2019.

[18] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 1–10. IEEE, 2009.

[19] Steven Bird and Edward Loper. Nltk: The natural language toolkit. In *Proceedings of the ACL 2004 on Interactive Poster and Demonstration Sessions*, ACLdemo '04, Stroudsburg, PA, USA, 2004. Association for Computational Linguistics.

[20] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 242–253, New York, NY, USA, 2018. ACM.

[21] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[22] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. Fuzzing: Challenges and reflections. *IEEE Software*, 2020.

[23] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[24] Nghi DQ Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36, 2018.

[25] David Bushnell. Jpf for beginners. `http://javapathfinder.sourceforge.net/events/JPF-workshop-050108/tutorial.pdf`, 2008. JPF Workshop 2008.

[26] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[27] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, December 2008.

[28] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[29] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. ICSE 2014, page 931–942, New York, NY, USA, 2014. Association for Computing Machinery.

[30] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 282–293. ACM, 2016.

[31] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, pages 385–400, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[32] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18. Springer, 2003.

[33] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. On the coherence between comments and implementations in source code. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 76–83. IEEE, 2015.

[34] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[35] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 357–366. IEEE, 2011.

[36] Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398, 2019.

[37] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer, 2008.

[38] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of {TLS} implementations. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 193–206, 2015.

[39] Sergio Cozzetti B de Souza, Nicolas Anquetil, and Káthia M de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.

[40] Luca Della Toffola, Cristian-Alexandru Staicu, and Michael Pradel. Saying 'hi!'is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49. IEEE, 2017.

[41] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability*, 30(3):e1733, 2020.

[42] M Eddington. Peach fuzzing platform. `http://peachfuzzer.com/FrontPage`.

[43] Arni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. *BRICS, Department of Computer Science, University of Aarhus, Denmark*, page 17, 2008.

[44] Elasticsearch. *Elasticsearch. Commit 3f2a241*, 2018 (July 4, 2018).

[45] Elasticsearch. *Elasticsearch. Commit 1a5e72e*, 2019 (February 26, 2019).

[46] Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering &#38; Applications Symposium*, IDEAS '15, pages 202–203, New York, NY, USA, 2014. ACM.

[47] Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79. IEEE, 2007.

[48] Gordon Fraser and Andrea 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*, pages 416–419. ACM, 2011.

[49] Gordon Fraser and Andrea 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*, ESEC/FSE '11, pages 416–419, 2011.

[50] Glide. *Glide. Commit ed20643fb*, 2018 (September 6, 2018).

[51] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[52] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[53] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[54] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 72–82, 2014.

[55] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. ICSE 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[56] M. Grechanik, K. M. Conroy, and K. A. Probst. Finding relevant applications for prototyping. In *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, pages 12–12, May 2007.

[57] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 475–484, New York, NY, USA, 2010. ACM.

[58] Guava. *Guava. Commit 3dfee64*, 2018 (December 6, 2018).

[59] Guava. *Guava. Commit af3ee1c*, 2018 (October 27, 2018).

[60] Guava. *Guava. Commit 7fdf1a6*, 2019 (October 3, 2019).

[61] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 182–191. IEEE, 2010.

[62] Yoshiki Higo and Shinji Kusumoto. How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 222–231. IEEE, 2012.

[63] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery: A case study of database systems. In *2007 IEEE International Conference on Software Maintenance*, pages 285–294, 2007.

[64] Pieter Hooimeijer and Margus Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 248–262. Springer, 2011.

[65] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. *SIGPLAN Not.*, 44(6):188–198, June 2009.

[66] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Elsevier, 1971.

[67] Roya Hosseini and Peter Brusilovsky. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, volume 1009, pages 60–63. University of Pittsburgh, 2013.

[68] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1):418–451, 2018.

[69] Werner Janjic and Colin Atkinson. Utilizing software reuse experience for automated test recommendation. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 100–106. IEEE, 2013.

[70] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. Context-based detection of clone-related bugs. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 55–64, New York, NY, USA, 2007. ACM.

[71] Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, and Martin Feilkas. Static bug detection through analysis of inconsistent clones. In *Software Engineering (Workshops)*, pages 443–446, 2008.

[72] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Mudablue: an automatic categorization system for open source repositories. In *11th Asia-Pacific Software Engineering Conference*, pages 184–193, Nov 2004.

[73] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116. ACM, 2009.

[74] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. Automatic and lightweight grammar generation for fuzz testing. *Computers & Security*, 36:1–11, 2013.

[75] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[76] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[77] Nuo Li, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 515–519, Washington, DC, USA, 2009. IEEE Computer Society.

[78] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[79] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[80] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. An efficient smt solver for string constraints. *Form. Methods Syst. Des.*, 48(3):206–234, June 2016.

[81] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[82] Hui Liu and Hee Beng Kuan Tan. Covering code behavior on input validation in functional testing. *Information and Software Technology*, 51(2):546–553, 2009.

[83] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, 2005.

[84] Pablo Loyola, Matt Staats, In-Young Ko, and Gregg Rothermel. Dodona: automated oracle data set selection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 193–203. ACM, 2014.

[85] Tim Finin James Mayfield Lushan Han, Abhay L Kashyap and Johnathan Weese. Umbc ebiquity-core: Semantic textual similarity systems. In *Proceedings of the Second Joint Conference on Lexical and Computational Semantics*. Association for Computational Linguistics, June 2013.

[86] Paul W McBurney and Collin McMillan. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2):103–119, 2015.

[87] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[88] Collin McMillan, Negar Hariri, Denys Poshyvanyk, Jane Cleland-Huang, and Bamshad Mobasher. Recommending source code for use in rapid software prototypes. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 848–858, Piscataway, NJ, USA, 2012. IEEE Press.

[89] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[90] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 435–448, 2020.

[91] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[92] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[93] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, page 432–441, New York, NY, USA, 2005. Association for Computing Machinery.

[94] Vishal Misra, Jakku Sai Krupa Reddy, and Sridhar Chimalakonda. Is there a correlation between code comments and issues? an exploratory study. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 110–117, 2020.

[95] Stefano Mizzaro. Relevance: The whole history. *Journal of the American Society for Information Science*, 48(9):810–832.

[96] Stefano Mizzaro. How many relevances in information retrieval? *Interacting with computers*, 10(3):303–320, 1998.

[97] Anders Møller. dk. brics. automaton–finite-state automata and regular expressions for java, 2010, 2014.

[98] MPAndroidchart. *MPAndroidchart. Commit fcc5af71*, 2020 (January 22, 2020).

[99] mVerify Corporation. What's my testing roi". 2005. An mVerify Whitepaper.Chicago, Illinois.

[100] Netty. *Netty. Commit 8f01259*, 2018 (June 26, 2018).

[101] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522, 2016.

[102] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 457–468, New York, NY, USA, 2014. ACM.

[103] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830. IEEE, 2019.

[104] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar Al-Kofahi, and Tien N Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324, 2010.

[105] Octoverse, GitHub. Github. the state of the octoverse 2019., 2019. Retrieved September 30, 2019.

[106] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[107] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[108] Michael Petito. Eclipse refactoring. *http://people. clarkson. edu/˜ dhou/courses/EE564-s07/Eclipse-Refactoring. pdf*, 5:2010, 2007.

[109] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 124–145. Springer, 2006.

[110] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[111] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE, 2014.

[112] Achyudh Ram, Anand Ashok Sawant, Marco Castelluccio, and Alberto Bacchelli. What makes a code change easier to review: an empirical investigation on code change reviewability. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 201–212, 2018.

[113] Justus J Randolph. Free-marginal multirater kappa (multirater k [free]): An alternative to fleiss' fixed-marginal multirater kappa. *Online submission*, 2005.

[114] Justus J Randolph. Online kappa calculator. *Retrieved October*, 20:2011, 2008.

[115] Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 112–122. IEEE, 2017.

[116] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 419–428, 2014.

[117] ReactiveX. *RxJava. Commit 6ba932c*, 2019 (December 18, 2019).

[118] Soumaya Rebai, Oussama Ben Sghaier, Vahid Alizadeh, Marouane Kessentini, and Meriem Chater. Interactive refactoring documentation bot. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 152–162. IEEE, 2019.

[119] R Rehurek and P Sojka. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2), 2011.

[120] Talia Ringer, Dan Grossman, Daniel Schwartz-Narbonne, and Serdar Tasiran. A solver-aided language for test input generation. *Proc. ACM Program. Lang.*, 1(OOPSLA):91:1–91:24, October 2017.

[121] Romain Robbes and Michele Lanza. How program history can improve code completion. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 317–326. IEEE, 2008.

[122] Martin P Robillard and Robert Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[123] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, 2016.

[124] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 65–71, New York, NY, USA, 2006. ACM.

[125] Anita Sarma, Gerald Bortis, and André Van Der Hoek. Towards supporting awareness of indirect conflicts across software configuration management workspaces. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 94–103, 2007.

[126] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. Preventing input validation vulnerabilities in web applications through automated type analysis. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 233–243. IEEE, 2012.

[127] Dominic Seiffert and Oliver Hummel. Adapting collections and arrays: Another step towards the automated adaptation of object ensembles. *Journal of Systems and Software*, 123:79–91, 2017.

[128] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[129] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automated discovery of valid test strings from the web using dynamic regular expressions collation and natural language processing. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 79–88. IEEE, 2012.

[130] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Science of Computer Programming*, 97:405–425, 2015.

[131] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, 2001.

[132] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 13–22. IEEE, 2007.

[133] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[134] Grigori Sidorov, Alexander Gelbukh, Helena Gómez-Adorno, and David Pinto. Soft similarity and soft cosine measure: Similarity of features in vector space model. *Computación y Sistemas*, 18(3):491–504, 2014.

[135] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub, oct. de*, 2017.

[136] Spring-projects. *Spring-framework. SPR-16130*, 2017 (October 30, 2017).

[137] Spring-projects. *Spring-framework. Commit 859923b*, 2019 (June 11, 2019).

[138] Spring-projects. *Spring-boot. Commit a63ab46*, 2020 (April 28, 2020).

[139] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52, 2010.

[140] Kathryn T Stolee, Sebastian Elbaum, and Matthew B Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *Journal of Systems and Software*, 116:35–48, 2016.

[141] L. Tan, Y. Zhou, and Y. Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 11–20, May 2011.

[142] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[143] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.

[144] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, April 2012.

[145] Kunal Taneja, Nuo Li, Madhuri R. Marri, Tao Xie, and Nikolai Tillmann. Mitv: Multiple-implementation testing of user-input validators for web applications. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 131–134, New York, NY, USA, 2010. ACM.

[146] C. Teyton, J. Falleri, and X. Blanc. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 192–201, Oct 2013.

[147] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, New York, NY, USA, 2014. ACM.

[148] Shivkumar Hasmukhrai Trivedi. Software testing technique. *International Journal of Advanced Research in Computer Science and Software Engg 2 (10)*, pages 433–438.

[149] Gias Uddin and Martin P Robillard. How api documentation fails. *IEEE Software*, 32(4):68–75, 2015.

[150] Anthony J Viera, Joanne M Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.

[151] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[152] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. Bugram: Bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 708–719, New York, NY, USA, 2016. Association for Computing Machinery.

[153] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE, 2019.

[154] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 620–631, Piscataway, NJ, USA, 2015. IEEE Press.

[155] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 257–266, New York, NY, USA, 2010. ACM.

[156] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*, pages 1070–1076. IEEE, 2012.

[157] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157. Springer, 2010.

[158] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[159] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[160] Michal Zalewski. American fuzzy lop. *URL: http://lcamtuf. coredump. cx/afl*, 2017.

[161] Tianyi Zhang and Miryung Kim. Automated transplantation and differential testing for clones. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 665–676. IEEE, 2017.

[162] Y. Zhang, D. Lo, P. S. Kochhar, X. Xia, Q. Li, and J. Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23, Feb 2017.

[163] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.

[164] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204, 2010.

[165] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 27–37. IEEE Press, 2017.

[166] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

[167] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.