

# HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

A Dissertation Outline Presented

by

MANISH MOTWANI

Submitted to the Graduate School of the  
University of Massachusetts Amherst in partial fulfillment  
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2021

College of Information and Computer Sciences

© Copyright by Manish Motwani 2019

All Rights Reserved

# HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

A Dissertation Outline Presented

by

MANISH MOTWANI

Approved as to style and content by:

---

Yuriy Brun, Chair

---

Claire Le Goues, Member

---

Arjun Guha, Member

---

George S. Avrunin, Member

---

James Allan, Chair of the Faculty  
College of Information and Computer Sciences

# ABSTRACT

## HIGH-QUALITY AUTOMATIC PROGRAM REPAIR

MAY 2021

MANISH MOTWANI

B.S., IIIT HYDERABAD

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Yuriy Brun

A significant fraction of developers' time and effort is spent on fixing defects in the software. Automatic Program Repair is a research area that has recently gained attention because it proposes techniques to fix software defects with minimal or without any human intervention. While existing repair techniques can fix a large number of defects in real-world software, most of the repairs produced are not *correct* or acceptable to developers.

Whether manual or automated, software repair is essentially a three step process. The first step involves identifying program elements that cause the defect (fault localization). The second step involves identifying modifications to the defective program elements which would fix the defect (patch generation). Finally, the third step involves verifying that the modified program actually fixes the defect (patch validation). Recent studies have shown that using information about the defect from various sources including natural-language software artifacts such as bug reports can significantly improve automated fault localization. Similarly, using artifacts such as software specifications and developer-written code comments can improve developer-written tests which are typically used to constraint the search

space of the candidate patches (during patch generation) and to verify the correctness of the produced repair (during patch validation) by automated program repair techniques.

In this dissertation, we first describe how to objectively evaluate repair techniques along the dimensions of repair quality and repair applicability, and present the evaluation of these techniques for each dimension on real-world defects. We then propose multiple methods to improve fault localization and patch validation steps of the program repair process to improve the quality of the repairs produced. The proposed methods use machine learning techniques to transform relevant information extracted from different information sources into machine-processable form, and equip program repair techniques with this additional information during the repair process. With improved fault localization and patch validation, the search space of candidate patches could be more constrained which would likely improve the quality of repairs produced.

# TABLE OF CONTENTS

	Page
<b>ABSTRACT</b> .....	<b>iv</b>
<b>LIST OF TABLES</b> .....	<b>ix</b>
<b>LIST OF FIGURES</b> .....	<b>x</b>
<b>CHAPTER</b>	
<b>INTRODUCTION</b> .....	<b>1</b>
<b>1. AUTOMATED PROGRAM REPAIR</b> .....	<b>4</b>
1.1 Introduction .....	4
1.2 Automated Program Repair Process .....	4
1.3 Types of Program Repair Techniques .....	5
1.3.1 Search-based Repair Techniques .....	6
1.3.2 Semantics-based Repair Techniques .....	7
1.4 Open Challenges .....	8
<b>2. PROBLEMS IN AUTOMATED PROGRAM REPAIR</b> .....	<b>9</b>
2.1 Introduction .....	9
2.2 Applicability of Program Repair Techniques .....	10
2.2.1 Approach .....	10
2.2.2 Research Questions .....	10
2.2.3 Findings .....	11
2.2.4 Contributions .....	11
2.3 Quality of Generate-and-Validate Repair Techniques .....	12
2.3.1 Approach .....	13
2.3.2 Research Questions .....	13

2.3.3	Findings .....	14
2.3.4	Contributions .....	15
2.4	Quality of Semantics-Based Repair Techniques .....	15
2.4.1	Approach .....	16
2.4.2	Findings .....	16
2.4.3	Contributions .....	17
2.5	Key Findings and Research Question .....	19
<b>3.</b>	<b>IMPROVING PATCH VALIDATION .....</b>	<b>20</b>
3.1	Introduction .....	20
3.2	Proposed Approach .....	22
3.3	Evaluation .....	24
3.4	Comparison with the state-of-the-art test generation tools .....	25
3.5	Contributions .....	26
<b>4.</b>	<b>IMPROVING FAULT LOCALIZATION .....</b>	<b>28</b>
4.1	Introduction .....	28
4.2	Proposed Approach .....	29
4.2.1	Multi-layer Perceptron .....	29
4.2.2	Approach .....	30
4.3	Evaluation .....	32
4.3.1	Dataset .....	32
4.3.2	Evaluation metrics .....	33
<b>5.</b>	<b>HIGH QUALITY REPAIR USING IMPROVED FAULT LOCALIZATION AND PATCH VALIDATION .....</b>	<b>34</b>
5.1	Introduction .....	34
5.2	Proposed Method 1: Extend Java Repair Framework to incorporate improved FL .....	34
5.2.1	Dataset .....	35
5.2.2	Evaluation .....	36
5.3	Proposed Method 2: Neural Network based End-to-End Program Repair .....	36
5.3.1	Problem Definition .....	37
5.3.2	NPR Approach .....	38
5.3.3	Dataset .....	39

5.3.4	Evaluation .....	39
<b>6.</b>	<b>RELATED WORK .....</b>	<b>41</b>
6.1	Automatic Program Repair .....	41
6.2	Empirical Studies Evaluating Automatic Program Repair .....	44
6.3	Automated Fault Localization .....	46
6.4	Automated Test Generation .....	47
<b>7.</b>	<b>RESEARCH PLAN .....</b>	<b>50</b>
7.1	Projects and Timeline .....	50
7.2	Potential Risks and Contingency Plan .....	53
<b>8.</b>	<b>CONTRIBUTIONS .....</b>	<b>54</b>
 <b>APPENDIX: INFORMATION RETRIEVAL BASED FAULT LOCALIZATION .....</b>		 <b>55</b>
 <b>BIBLIOGRAPHY .....</b>		 <b>75</b>



## LIST OF TABLES

Table	Page
2.1 Comparison of average patch-quality of SOSRepair(SOS) and SOSRepair provided with fault location (SOS+) with other state-of-the-art repair techniques. ....	17
5.1 The 357 defect dataset created from five real-world projects in the Defects4J version 1.1.0 benchmark. We used <code>SLOCCount</code> to measure the lines of code (KLoC) counts ( <a href="https://www.dwheeler.com/sloccount/">https://www.dwheeler.com/sloccount/</a> ). The <code>tests</code> and <code>test KLoC</code> columns refer to the developer-written tests. ....	35

## LIST OF FIGURES

Figure	Page
1.1 Three-step process of program repair. The first step ( <i>fault localization</i> ) involves identifying source code elements that could be potentially buggy, the second step ( <i>patch generation</i> ) involves producing a patch (program modification) that can be applied to the buggy part of the program, and the third step ( <i>patch validation</i> ) involves patching the buggy program with a candidate patch and validating it against the test suite. If all tests pass, the patched program is reported as a repair and the process terminates otherwise, an attempt is made to produce new patch until the search space is exhausted or a timeout occurs. . . . .	5
3.1 Section 15.4.2.2 of ECMA-262 (v5.1), specifying the JavaScript <code>Array(len)</code> constructor. . . . .	21
3.2 Swami generates tests by applying a series of regular expressions to processed (e.g., tagged with parts of speech) natural language specifications, discarding non-matching sentences. Swami does not require access to the source code; however, Swami can optionally use the code to identify relevant specifications. Swami’s output is executable tests with oracles and test templates that can be instantiated to generate more tests. . . . .	22
3.3 The executable tests automatically generated for the <code>Array(len)</code> constructor from the specification in Figure 3.1. . . . .	23
3.4 ECMA specifications include references to abstract operations, which are formally defined elsewhere in the specification document, but have no public interface. Section 9.6 of ECMA-262 (v5.1) specifies the abstract operation <code>ToUnit32</code> , referenced in the specification in Figure 3.1. . . . .	24
4.1 A Multi-Layer Perceptron with one input layer, one hidden layer and one output layer and the definitions for four widely used activation functions in neural networks. . . . .	30

4.2	The proposed MLP architecture to combine 10 different fault localization techniques which belong to six different classes. The input to the model is a $1 \times 10$ dimension <i>feature vector</i> that captures the suspiciousness scores of a source code statement computed using 10 fault localization techniques. The output of the model is a $1 \times 2$ dimension vector that contains the probabilities of the input to be buggy and not buggy. . . . .	32
5.1	The proposed end-to-end neural network-based program repair technique based on sequence-to-sequence neural machine translation(NMT) technique. During training, the model is fed pair of buggy source code and human-written patch. During testing, the model takes as input buggy source code and determines buggy program elements using NFL. Next, it uses the trained model to predict the candidate patches for the buggy program elements. Finally, the candidate patches are evaluated using provided test-suite to validate their correctness. . . . .	40
7.1	Proposed timeline for this dissertation. Top chart shows the lists of all the projects that are pursued as part of this dissertation along which their timeline. The green color indicates the projects that are completed and the orange indicates the projects/tasks that are work in progress. Bottom chart shows the detailed timeline plan for completing the projects which are work in progress and the task of writing the dissertation. . . . .	52
A.1	Specifications for Array join and Array pop operations from <i>ECMA-262-v3</i> represented in XML format . . . . .	60
A.2	Natural language concern from <i>ECMA-262-v3</i> describing the implementation of Array.join operation in Rhino . . . . .	61
A.3	Array.join operation implemented in NativeArray.java file of Rhino Version 1.5R6 source code . . . . .	61
A.4	IRFL Architecture . . . . .	63
A.5	Performance of IRFL after tuning model parameters for the Rhino dataset. The optimal values of model parameters were found to be $k_1 = 0.5$ and $b = 0.6$ . . . . .	70
A.6	IRFL 's performance considering top N ( $N \in \{1, 5, 10, 10000\}$ ) documents retrieved per concern. . . . .	71
A.7	Distribution of precision per query (PPQ) and recall per query (RPQ) considering all the documents retrieved for the 279 Rhino concerns of the ground-truth dataset. . . . .	73

## INTRODUCTION

The global cost of software debugging has risen to \$312 billion annually, and a significant amount of developers' time is spent on debugging and repairing software defects [58, 186]. Automated program repair research (e.g., [16, 27, 29, 36, 37, 77, 86, 87, 88, 92, 103, 106, 107, 112, 147, 168, 172, 173, 193, 195]) aims to address this problem by devising techniques that can automatically produce software patches to fix defects with minimal or without requiring any human intervention. For example, Facebook uses two automated program repair tools, SapFix and Getafix, in their production pipeline to suggest bug fixes [108, 157]. The goal of automated program repair techniques is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite. Unfortunately, these patches can repair some functionality encoded by the tests, while simultaneously breaking other, undertested functionality [163]. Thus, *quality* of the resulting patches is a critical concern. Recent results suggest that patch overfitting — patches that pass a particular set of test cases supplied to the program repair tool but fail to generalize to the desired specification — is common [89, 105, 137, 163].

Most of the state-of-the-art program repair techniques use developer-written tests to: (a) localize the defect typically using spectrum-based fault localization techniques which use the runtime information of passing and failing tests to localize the defective program elements, and (b) generate and validate the automatically produced candidate patches based on the constraints imposed by the tests. While test suites provide an easy-to-use (because they are executable) specification, software typically contains many more artifacts that describe the desired *correct* software behaviour. Many of these artifacts such as requirements specifications, code comments, and bug reports use natural-language text to describe the bug and

intended software behavior, and are therefore not directly used in the repair process. We hypothesize that if we can derive executable constraints from such artifacts and equip repair techniques with these additional constraints, it could further constraint the search space of the candidate patches and would improve the quality of patches produced. The central goal of this work is to test this hypothesis.

We propose a natural-language processing based approach that can automatically extract executable tests with precise oracles from structured natural language software specifications. The generated tests cover code which are previously not covered by the developer-written tests. The developer-written tests when augmented with these automatically generated tests would sharpen the constraints on the search space of the candidate patches and therefore will improve the patch generation and patch validation steps. We also propose a method to improve the fault localization by using machine learning techniques to combine 10 different fault localization techniques that use six different sources of information about defect to identify the program elements that cause that defect. Finally, we propose multiple ways in which we can integrate the improved fault localization and patch validation in the automated program repair process.

The rest of this proposal is structured as follows. Chapter 1 describes the background on automated program repair, the program repair process, and open challenges in the automated program repair research area. Chapter 2 describes ways to objectively evaluate the repair techniques along the dimensions of repair applicability and repair quality, presents the evaluation of program repair techniques along these dimensions on real-world defects, and describes the key findings and research problem that motivate this dissertation. Chapter 3 proposes a method to automatically generate executable tests from structured natural-language software specifications. Chapter 4 proposes a method to combine multiple fault localization techniques to improve the fault localization. Chapter 5 describes two methods to integrate the improved fault-localization and patch validation in automated program repair process. Chapter 6 places this work in the context of related research. Chapter 7 describes

the research plan with timeline and potential risks along with contingency plan. Chapter 8 summarizes proposed contributions.

# CHAPTER 1

## AUTOMATED PROGRAM REPAIR

### 1.1 Introduction

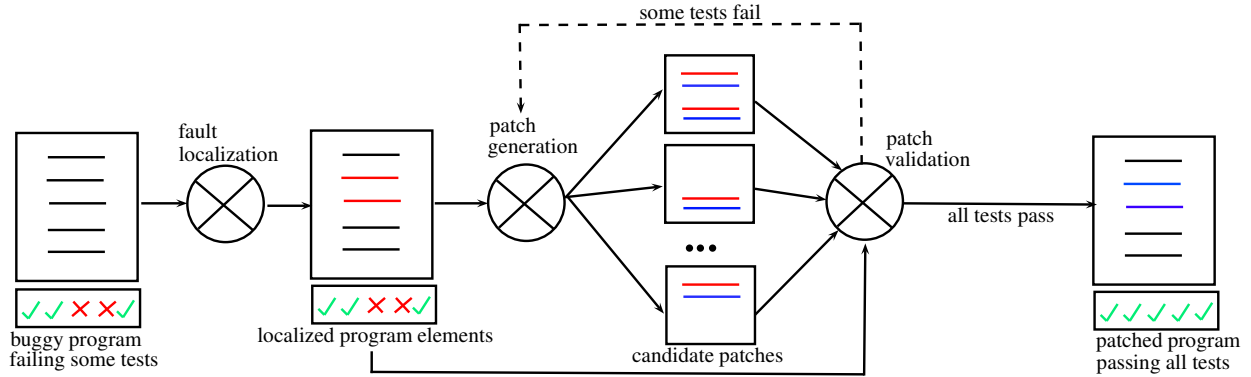
The goal of automated program repair techniques is to take a program and a suite of tests, some of which that program passes and some of which it fails, and to produce a patch that makes the program pass all the tests in that suite. This chapter describes some background on automated program repair techniques. The remaining of this chapter is organized as follows. Section 1.2 describes the program repair process, Section 1.3 describes types of program repair techniques, and Section 1.4 lists the current open challenges in the field of automated program repair.

### 1.2 Automated Program Repair Process

Figure 1.1 shows the high-level program repair process which can be broken down into following three steps.

1. **Fault localization:** The goal of fault localization is to identify defective program elements that cause the software defect. Automated fault localization typically uses static and run-time information about the program to identify program elements that may be the root cause of the defect. Chapter 4 describes this in more detail.
2. **Patch generation:** This is the core algorithm of the program repair technique. Different techniques use different algorithms to produce patches to fix the defect. Section 1.3 describes these techniques in more detail.

3. **Patch validation:** This involves modifying a buggy program by applying the automatically produced patch and validating its correctness against the developer-written test suite. If the patched program passes all the tests, the corresponding patch is reported as a *plausible* patch (patch that passes at least all tests used in the repair process). It is important to note that a plausible patch may overfit to the tests and hence may not necessarily be a *correct* patch. Chapter 2 describes this in more detail.



**Figure 1.1.** Three-step process of program repair. The first step (*fault localization*) involves identifying source code elements that could be potentially buggy, the second step (*patch generation*) involves producing a patch (program modification) that can be applied to the buggy part of the program, and the third step (*patch validation*) involves patching the buggy program with a candidate patch and validating it against the test suite. If all tests pass, the patched program is reported as a repair and the process terminates otherwise, an attempt is made to produce new patch until the search space is exhausted or a timeout occurs.

### 1.3 Types of Program Repair Techniques

Based on the underlying repair approach, program repair techniques can be classified into two categories: *search-based* techniques and *semantics-based* techniques. The ideas proposed in this dissertation are applicable to both classes of techniques.

#### 1.3.1 Search-based Repair Techniques

*Search-based* techniques use certain heuristics or predefined templates to generate many syntactic candidate patches, validating them against the tests (e.g., GenProg [92], Prophet



[106], AE [183], HDRRepair [88], ErrDoc [172], JAID [27], Qlose [36], and Par [77], among others).

Techniques such as DeepFix [62] and ELIXIR [153] use learned models to predict erroneous program locations along with patches. ssFix [193] uses existing code that is syntactically related to the context of a defect to produce patches. CapGen [187] works at the AST node level (token-level) and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. To manage the large search space of candidates created because of using finer-level granularity, it extracts context information from candidate code snippets and prioritizes the mutation operators considering the extracted context information. SimFix [71] considers the variable name and method name similarity in addition to the structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. Hercules [154] generalizes single-location program repair techniques to defects that require similar edits be made in multiple locations. Enforcing that a patch keeps a program semantically similar to the buggy version by ensuring that user-specified correct traces execute properly on the patched version can repair reactive programs with linear temporal logic specifications [174].

Several repair approaches have aimed to reduce syntactic or semantic differences between the buggy and patched program e.g., [36, 71, 76, 87, 111, 174, 187], with a goal of improving patch quality. For example, Qlose [36] minimizes a combination of syntactic and semantic differences between the buggy and patched programs while generating candidate patches. SketchFix [68] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution. SOFix [98] uses 13 predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions

and answers for fine-grained modifications. SapFix [108] and Getafix [157], two tools deployed on production code at Facebook, efficiently produce repairs for large real-world programs. SapFix [108] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in realtime. Getafix [157] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook’s in-house static analysis tool.

### 1.3.2 Semantics-based Repair Techniques

*Semantics-based* techniques use constraint solving and program synthesis to synthesize patches to satisfy semantics constraints extracted via symbolic execution and provided test suites (e.g., Nopol [195], Semfix [124], DirectFix [111], Angelix [112], S3 [87], JFIX [86]). SemGraft [110] infers specifications by symbolically analyzing a correct reference implementation (as opposed to using test cases). Genesis [103], Refazer [147], NoFAQ [37], Sarfgen [178], and Clara [61] process correct patches to automatically infer code transformations to generate patches, a problem conceptually related to our challenge in integrating repair snippets to a new context.

SearchRepair [76] combines these classes, using a constraint solver to identify existing code to construct repairs. SOSRepair [5], builds on SearchRepair by fundamentally improving the approach in several important ways. It is significantly more expressive (handling code constructs used in real code and reasoning about snippets that can affect multiple variables as output) and scalable (SearchRepair could only handle small, student-written C programs), supports deletion and insertion, uses failing test cases to restrict the search space, repairs code without passing examples, and its encoding of the repair query is significantly more expressive and efficient.

## 1.4 Open Challenges

This section describes the current open challenges in the automated program repair research area identified in recent studies [54, 59]. Following are the three high-level open challenges that summarize various challenges listed in these works.

1. **Fix Correctness/Quality Challenge:** This involves increasing the chance that a repair technique provides a correct fix that is easy to understand and maintain in the long term. Addressing this challenge is perhaps the most important step toward real-life adoption of program repair. Although it might be infeasible to produce fixes that are guaranteed to satisfy the developers expectations, finding methods to generate and evaluate fixes that are likely acceptable by developers is an open problem.
2. **Scope/Technical Challenge:** This involves extending the applicability of the state-of-the-art repair techniques to the kinds of bugs which it cannot repair. There are various aspects to addressing this challenge such as how to evaluate the applicability given so many independent studies of multiple repair techniques which are still scattered and difficult to be synthesized into a clear picture, and how to improve the design of the repair techniques to enable them to fix more complex defects.
3. **Process Challenge:** This involves integrating repair tools into development processes ensuring reliability and without disrupting the development process significantly. This includes challenges such as how to design repair techniques that fix programs without negatively affecting the maintainability of software, and how to integrate repair tools with defect detection tools such as compilers or integrated development environments' (IDEs) debugging components.

In this dissertation, we propose multiple method to address the *Fix Correctness/Quality Challenge* and one of the aspects of *Scope/Technical Challenge*.

## CHAPTER 2

### PROBLEMS IN AUTOMATED PROGRAM REPAIR

#### 2.1 Introduction

Existing evaluations of automated repair techniques focus on the fraction of the defects for which the technique can produce a patch, the time needed to produce patches, and how well patches generalize to the intended specification. However, all these evaluations use different benchmarks and techniques which prevents us from determining the clear picture of whether automated repair techniques are capable of repairing defects that developers consider important or that are hard for developers to repair manually. Further, the existing evaluations of patch correctness use manual inspection of automatically generated patches which is likely to be biased. In this dissertation, we propose an objective evaluation framework to evaluate repair techniques along the dimensions of repair applicability and repair quality. We perform a large scale evaluation of the state-of-the-art repair techniques using our evaluation framework on real-world defects.

The remaining of this chapter is organized as follows. Section 2.2 describes the objective way to evaluate the applicability of repair techniques and the key findings from this study, Section 2.3 describes the objective way to evaluate the quality of repair techniques and the evaluation of search-based (generate-and-validate) repair techniques, Section 2.4 describes the key findings of evaluating the quality of SOSRepair, a novel semantics-based repair technique which builds upon an existing technique that is able to produce high-quality patches, and Section 2.5 describes the key observations reconciled from all these evaluations that motivates us to pose the research question we wish to investigate in this dissertation.

## 2.2 Applicability of Program Repair Techniques

Automated program repair techniques use a defective program and a partial specification (typically a test suite) to produce a patch (program modification) which when applied to the defective program, it produces a program variant that satisfies the specification. While prior work has studied patch quality [137, 163] and maintainability [52], it has not examined whether automated repair techniques are capable of repairing defects that developers consider important or that are hard for developers to repair manually. We perform a study to tackle such questions.

### 2.2.1 Approach

Our study considers nine automated repair techniques for C and Java: AE [183], GenProg [185], a Java reimplementaion of GenProg [109], Kali [137], a Java reimplementaion of Kali [109], Nopol [38], Prophet [106], SPR [104], and TrpAutoRepair [136].

We analyze popular bug tracking systems and source code repositories to identify parameters relevant to **defect importance**, **independence**, and **complexity**, and **test effectiveness**. We compute these parameters for two benchmarks of defects often used to evaluate automated program repair, ManyBugs [91] (185 C defects) and Defects4J [74] (357 Java defects). We further analyze developer-written patches for these defects to identify characteristics of those patches that may influence automated repair. Finally, to evaluate the applicability of repair techniques, we run statistical tests to compute the association between these parameters and the ability to repair defects of repair techniques.

### 2.2.2 Research Questions

Our study answers the following questions: Is a repair technique’s ability to produce a patch for a defect correlated with that defect’s (RQ1) importance, (RQ2) complexity, (RQ3) effectiveness of the test suite, or (RQ4) dependence on other defects? (RQ5) What characteristics of the developer-written patch are significantly associated with a repair tech-

nique’s ability to produce a patch? And, (RQ6) what defect characteristics are significantly associated with a repair technique’s ability to produce a high-quality patch?

### 2.2.3 Findings

We find that (RQ1) Java repair techniques are moderately more likely to patch higher-priority defects; for C, there is no correlation. There is little to no consistent correlation between producing a patch and the time taken by developer(s) to fix the defect, as well as the number of software versions affected by that defect. (RQ2) C repair techniques are less likely to patch defects that required developers to write more lines of code and edit more files. (RQ3) Java repair techniques are less likely to patch defects with more triggering or more relevant tests. Test suite statement coverage has little to no consistent correlation with producing a patch. (RQ4) Java repair techniques’ ability to patch a defect does not correlate with that defect’s dependence on other defects. (RQ5) Repair techniques struggle to produce patches for defects that required developers to insert loops or new function calls, or change method signatures. Finally, (RQ6) only two of the considered repair techniques, Prophet and SPR, produce a sufficient number of high-quality patches to evaluate. These techniques were less likely to patch more complex defects, and they were even less likely to patch them correctly.

### 2.2.4 Contributions

The main contributions of this study are:

- The publicly-released annotation of 409 defects in ManyBugs and Defects4J, to be used for evaluating automated repair applicability.
- A methodology for evaluating the applicability of repair techniques, with the goal of encouraging research to focus on important and hard defects.
- The evaluation of nine automated program repair techniques’ applicability to 409 ManyBugs and Defects4J defects.

The full version of this study [121] can be found at <http://dx.doi.org/10.1007/s10664-017-9550-0>.

## 2.3 Quality of Generate-and-Validate Repair Techniques

While it is less likely for repair techniques to fix hard or complex bugs as described in the findings of the previous section, even for the relatively less complex bugs, the quality of patches produced by many automated program repair techniques are often of low quality [163] and not semantically equivalent to developer-written patches [137]. This both raises an important concern about the practical usability of modern automated repair techniques, and drives research toward building techniques that produce higher-quality patches [76, 104, 106, 112].

Automated program repair techniques typically start with a program version and a set of passing and failing tests, and then modify the program version until finding a set of modifications (a patch) that makes all the tests pass. The underlying issue is that the set of tests provides a partial specification of the desired behavior, and thus the produced patches may overfit to those tests. For example, while, typically, many patches in a technique’s search space pass the supplied tests, relatively few are equivalent to the developer-written patch [106, 137]; the automated repair technique has no way of knowing which is the better patch to return.

Prior work introduced an objective methodology for evaluating the quality of a patch [163] and was successfully applied to a set of very small programs written by novice developers in an introductory programming course [163]. While that work identified important shortcomings of automated program repair techniques, its results may not generalize beyond the very small and simple programs. That study only considered two generate-and-validate repair techniques, did not control for confounding factors, and used test suite size as a proxy for coverage. By contrast, we perform a detailed large-scale study with four generate-and-validate repair techniques on 357 real-world defects in five real-world, large, complex projects

from the Defects4J benchmark [74] employing rigorous statistical analyses, properly measuring coverage, and controlling for confounding factors.

### 2.3.1 Approach

Our methodology for measuring patch quality relies on an independent test suite that is not given to the repair technique to produce a patch. The independent test suite captures (again, partially) some of the specifications not captured by the original test suite given to the repair technique, and thus its passing rate independently evaluates the quality of the patch. We generate these independent test suite using state-of-the-art automated test generation tools which can generate tests from a given source code. For the defect benchmarks that are used to evaluate repair techniques, we have a defective and developer-patched version of the program. We use human-patched version of the program to generate the independent test suite considering the developer-patched version as the oracle. Further, we ensure that independent test suite is of *high-quality* by making sure that it has 100% statement coverage on the buggy method (or developer-modified method) and atleast 80% statement covarege on the buggy class (or developer-modified class). The alternative to this methodology is a manual inspection of the patch, (e.g., [137]), but two independent recent studies [85, 197] have empirically demonstrated that our independent-test-suite-based methodology is more reliable and more objective than manual inspection.

### 2.3.2 Research Questions

Our study answers following research questions: (RQ1) Do generate-and-validate repair techniques produce patches for real-world defects?, (RQ2) How often and how much do the patches produced by generate-and-validate repair techniques overfit to the developer-written test suite and fail to generalize to the evaluation test suite, and thus ultimately to the program specification?, (RQ3) How do the coverage and size of the test suite used to produce the patch affect patch quality?, (RQ4) How does the number of tests that a buggy program fails affect the degree to which the generated patches overfit?, (RQ5) How does



the test suite provenance (whether it is written by developers or generated automatically) influence patch quality?, and (RQ6) Can overfitting be mitigated by exploiting randomness in the repair process? Do different random seeds overfit in different ways?

### 2.3.3 Findings

We find that (RQ1) Repair techniques produce patches for real-world defects although less often for Java defects than for C defects. (RQ2) Repair techniques often overfit to the developer-written test suite. Only between 13.8% and 41.6% of the patches pass 100% of an independent test suite. Comparing the patched version against the buggy version of the program reveals that patches typically break more functionality than they repair. (RQ3) Larger test suites produce slightly higher-quality patches, though, surprisingly, the effect is extremely small. Also surprisingly, there is no clear relationship between higher-coverage test suites and quality. (RQ4) More number of failing tests leads to slightly higher-quality patches. Spectrum-based fault localization gets significantly affected by the failing tests preventing some of the repair techniques to produce any patch. (RQ5) Test suite provenance has a significant effect on repair quality, although the effect may differ for different techniques. In most cases, human-written tests lead to higher-quality patches. (RQ6) The patches exhibit insufficient diversity to improve quality through some method of combining multiple patches.

Prior studies of quality of automated program repair have either used manual inspection for quality assessment [131,163,182], or have focused on small programs and relatively-easy-to-fix defects [163,197]. Some studies did use a 224-defect subset of the same benchmark of real-world programs we use, but used manual inspection for quality assessment and, unlike our work, assessed tools' ability to produce patches and efficiency of patch production, but did not identify the factors that affect patch quality (RQs 3–6) [42,109].

Employing the objective, independent-test-suite-based evaluation of patch quality, requires the creation of high-quality, automatically-generated test suites for real-world Java

projects. In this study, we develop a methodology for using today’s state-of-the-art test-suite generation techniques and overcoming their shortcomings to produce high-quality suites. We release both the methodology and the generated test suites.

### 2.3.4 Contributions

The main contributions of this study are:

- An empirical evaluation of quality of program repair on real-world Java defects, which outlines shortcomings and establishes a methodology and dataset for evaluating quality of new repair techniques’ patches on real-world defects to promote research on high-quality repair.
- A methodology for evaluating patch quality that fixes numerous shortcomings in prior work, properly controlling for potential confounding factors.
- A dataset of independent evaluation test suites for Defects4J defects, and a methodology for generating such test suites. Augmenting existing Defects4J defects with two, independently created test suites can aid not only program repair, but other test-based technology.
- Java Repair Framework (<http://JaRFly.cs.umass.edu/>), a publicly released, open-source framework for building Java generate-and-validate repair techniques, including our reimplementations of GenProg [90], Par [77], and TrpAutoRepair [136]. JaRFly allows for easy combinations and modifications to existing techniques, and simplifies experimental design for automated program repair on Java programs.

This work is under review in IEEE Transactions on Software Engineering (TSE).

## 2.4 Quality of Semantics-Based Repair Techniques

In this study, we also evaluate the quality of semantics-based repair techniques which use constraint solving and program synthesis to synthesize patches to satisfy semantics con-

straints extracted via symbolic execution and provided test suites (recall Chapter 1, Section 1.3).

### 2.4.1 Approach

We design a novel repair technique called SOSRepair which builds on the underlying principles of SearchRepair [76], a semantics-based repair technique which is able to produce high-quality patches (patches passed 97.3% of independent tests not used during patch construction) for bugs in small 24-line student-written programs but could not be run on large programs. We fundamentally redesign SearchRepair to create SOSRepair which is able to produce patches for real-world defects in large programs hypothesizing that SOSRepair would produce high-quality patches for large programs. We use our patch quality evaluation methodology described in Section 2.3 to evaluate the quality of the patches produced by SOSRepair for 65 real-world defects of 7 large open-source projects from the ManyBugs defect benchmark [91].

### 2.4.2 Findings

We found that SOSRepair was able to patch 22 (34%) of the 65 defects including 1 which was not patched by prior techniques however, the quality of the SOSRepair was comparable to prior techniques (Angelix [112], Prophet [106], and GenProg [92]) and was the same as obtained using SearchRepair on small programs. Table 2.1 shows the average patch-quality of SOSRepair (indicated by SOS) and SOSRepair provided with fault location (indicated by SOS+) compared to existing state-of-the-art repair techniques. The values describe the average of the fraction of independently generated evaluation test suites passed by the patched program for commonly patched defects by a given pair of techniques. We also found that fault localization is a key factor in SOSRepair’s success. Manually improving fault localization enabled SOSRepair to patch more defects and also produce more higher-quality patches.

Existing Technique	SOS	SOS+	#Commonly Patched Defects
Angelix (94.10)	84.22		9
Angelix (96.43)		99.29	7
Prophet (88.42)	81.50		11
Prophet (88.50)		93.42	12
GenProg (65.00)	88.00		16
GenProg (56.21)		95.50	14

**Table 2.1.** Comparison of average patch-quality of SOSRepair(SOS) and SOSRepair provided with fault location (SOS+) with other state-of-the-art repair techniques.

### 2.4.3 Contributions

To make SOSRepair possible, we make five major contributions to both semantic code search and program repair:

1. **A more-scalable semantic search query encoding.** We develop a novel, efficient, general mechanism for encoding semantic search queries for program repair, inspired by input-output component-based program synthesis [69]. This encoding efficiently maps the candidate fix code to the buggy context using a single query over an arbitrary number of tests. By contrast, SearchRepair [76] required multiple queries to cover all test profiles and failed to scale to large code databases or queries covering many possible permutations of variable mappings. Our new encoding approach provides a significant speedup over the prior approach, and we show that the speedup grows with query complexity.
2. **Expressive encoding capturing real-world program behavior.** To apply semantic search to real-world programs, we extend the state-of-the-art constraint encoding mechanism to handle real-world C language constructs and behavior, including structs, pointers, multiple output variable assignments, console output, loops, and library calls.

3. **Search for patches that insert and delete code.** Prior semantic-search-based repair could only *replace* buggy code with candidate fix code to affect repairs [76]. We extend the search technique to encode deletion and insertion.
4. **Automated, iterative search query refinement encoding negative behavior.** We extend the semantic search approach to include negative behavioral examples, making use of that additional information to refine queries. We also propose a novel, iterative, counter-example-guided search-query refinement approach to repair buggy regions that are not covered by the passing test cases. When our approach encounters candidate fix code that fails to repair the program, it generates new undesired behavior constraints from the new failing executions and refines the search query, reducing the search space. This improves on prior work, which could not repair buggy regions that no passing test cases execute [76].
5. **Evaluation and open-source implementation.** We implement and release SOSRepair (<https://github.com/squaresLab/SOSRepair>), which reifies the above mechanisms. We evaluate SOSRepair on the ManyBugs benchmark [91] commonly used in the assessment of automatic patch generation tools (e.g., [106, 112, 136, 183]). These programs are four orders of magnitude larger than the benchmarks previously used to evaluate semantic-search-based repair [76]. We show that, as compared to previous techniques applied to these benchmarks (Angelix [112], Prophet [106], and GenProg [92]), SOSRepair patches one defect none of those techniques patch, and produces patches of comparable quality to those techniques. We measure quality objectively, using independent test suites held out from patch generation [163]. We therefore also release independently-generated held-out test suites (<https://github.com/squaresLab/SOSRepair-Replication-Package>) for the defects we use to evaluate SOSRepair.

The full version of this study [5] can be found at <http://dx.doi.org/10.1109/TSE.2019.2944914>.

## 2.5 Key Findings and Research Question

Following are the key observations made from all of the above described evaluations of automated repair techniques.

1. Automated program repair techniques are less likely to patch more complex defects, and they are even less likely to patch them correctly (Section 2.2).
2. Repair techniques often overfit to the developer-written test suite. Patches typically break more functionality than they repair (Section 2.3).
3. Spectrum-based fault localization gets significantly affected by the failing tests preventing some repair techniques to produce any patch (Section 2.3).
4. Fault localization is a key factor in SOSRepair’s success. Manually improving fault localization enabled SOSRepair to patch more defects and also produce more higher-quality patches (Section 2.4).

These key findings suggest that fault localization and developer-written test suites used to guide the repair significantly affect the quality of the repair produced by program repair techniques. Based on this, we ask the following research question which directs the remainder of the research work proposed in this dissertation.

**Research Question:** Does improvement in fault localization and test-suites used to guide the repair process improve the quality of the patches produced by repair techniques?

## CHAPTER 3

### IMPROVING PATCH VALIDATION

#### 3.1 Introduction

One of the key steps in the program repair process is verifying that the patched program does what developer want it to do (recall Section 1.2 in Chapter 1). Unfortunately, the most common way humans describe and specify software is natural language, which is difficult to formalize, and thus also difficult to use in an automated process as an oracle of what the software should do. Hence, repair techniques use developer-written test suites as a proxy for software specification.

In this chapter, we propose an approach to automatically generate tests from natural language specifications which can be used to verify that the software does what the specifications say it should. Tests consist of two parts, an input to trigger a behavior and an oracle that indicates the expected behavior. Oracles encode intent and are traditionally specified manually, which is time consuming and error prone. While formal, mathematical specifications that can be used automatically by computers are rare, developers do write natural language specifications, often structured, as part of software requirements specification documents. For example, Figure 3.1 shows a structured, natural language specification of a JavaScript `Array(len)` constructor (part of the official JavaScript specification ECMA-262 standard [188]) to be implemented in JavaScript engines. The proposed approach focuses on generating oracles from such structured natural language specifications (test inputs can often be effectively generated randomly [50, 127], and together with the oracles, produce executable tests).

### 15.4.2.2 new Array (len)

The `[[Prototype]]` internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of `Array.prototype` (15.4.3.1). The `[[Class]]` internal property of the newly constructed object is set to `"Array"`. The `[[Extensible]]` internal property of the newly constructed object is set to `true`.

If the argument `len` is a Number and `ToUint32(len)` is equal to `len`, then the `length` property of the newly constructed object is set to `ToUint32(len)`. If the argument `len` is a Number and `ToUint32(len)` is not equal to `len`, a `RangeError` exception is thrown.

If the argument `len` is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to `len` with attributes `{[[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}`.

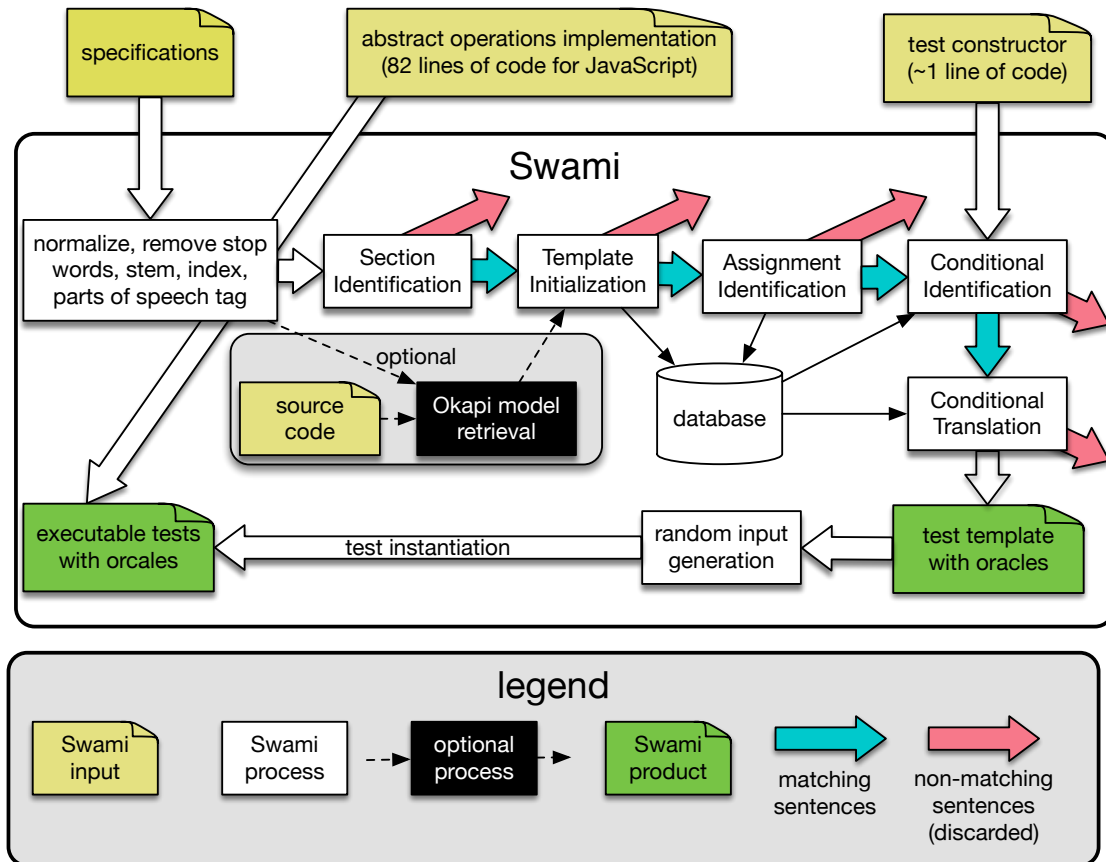
<https://www.ecma-international.org/ecma-262/5.1/#sec-15.4.2.2>

**Figure 3.1.** Section 15.4.2.2 of ECMA-262 (v5.1), specifying the JavaScript `Array(len)` constructor.

Of particular interest is generating tests for exceptional behavior and boundary conditions because, while developers spend significant time writing tests manually [9, 142], they often fail to write tests for such behavior. In a study of ten popular, well-tested, open-source projects, the coverage of exception handling statements lagged significantly behind overall statement coverage [57]. For example, Developers often focus on the common behavior when writing tests and forget to account for exceptional or boundary cases [9]. At the same time, exceptional behavior is an integral part of the software as important as the common behavior. An IBM study found that up to two thirds of production code may be devoted to exceptional behavior handling [32]. And exceptional behavior is often more complex (and thus more buggy) because anticipating all the ways things may go wrong, and recovering when things do go wrong, is inherently hard. Finally, exceptional behavior is often the cause of field failures [184], and thus warrants high-quality testing.

We present Swami, a technique for automatically generating executable tests from natural language specifications. We scope our work by focusing on exceptional and boundary behavior, precisely the important-in-the-field behavior developers often undertest [57, 184].





**Figure 3.2.** Swami generates tests by applying a series of regular expressions to processed (e.g., tagged with parts of speech) natural language specifications, discarding non-matching sentences. Swami does not require access to the source code; however, Swami can optionally use the code to identify relevant specifications. Swami’s output is executable tests with oracles and test templates that can be instantiated to generate more tests.

### 3.2 Proposed Approach

Figure 3.2 shows Swami approach. It takes as input a specification document that has hundreds of pages and specifications. Swami uses regular expressions to identify what sections of structured natural language specifications encode testable behavior. (While not required, if the source code is available, Swami can also use information retrieval techniques to identify such sections.) Swami then applies a series of four regular-expression-based rules to extract information about the syntax for the methods to be tested, the relevant variable assignments, and the conditionals that lead to visible oracle behavior, such as return state-

```

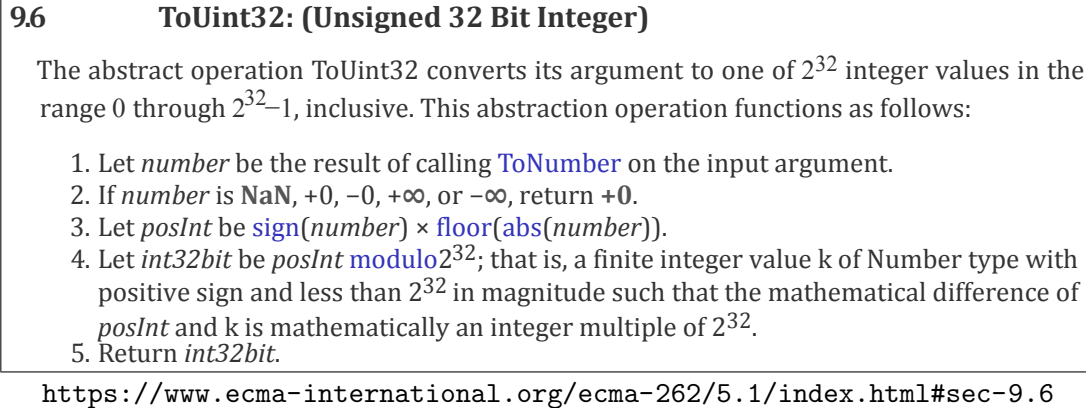
1  /*ABSTRACT FUNCTIONS*/
2  function ToUint32(argument){
3      var number = Number(argument)
4      if (Object.is(number, NaN) || number == 0 ||
5          number == Infinity || number == -Infinity ||
6          number == +0 || number == -0){
7          return 0
8      }
9      var i = Math.floor(Math.abs(number))
10     var int32bit = i%(Math.pow(2,32))
11     return int32bit
12 }
13 ...
14 /*TEST TEMPLATE GENERATED AUTOMATICALLY*/
15 function test_new_array(len){
16     if (typeof(len)!="number" && (ToUint32(len)!=len)){
17         try{
18             var output = new Array ( len );
19         }catch(e){
20             new TestCase("array_len", "array_len",
21                 true, eval(e instanceof RangeError))
22             test();
23         }
24     }
25 }
26 /*TESTS GENERATED AUTOMATICALLY*/
27 test_new_array(1.1825863363010669e+308);
28 test_new_array(null);
29 test_new_array(-747);
30 test_new_array(368);
31 test_new_array(false);
32 test_new_array(true);
33 test_new_array("V7K008H");
34 test_new_array(Infinity);
35 test_new_array(undefined);
36 test_new_array(/[\^.]+/);
37 test_new_array(+0);
38 test_new_array(NaN);
39 test_new_array(-0);
40 ...

```

**Figure 3.3.** The executable tests automatically generated for the `Array(len)` constructor from the specification in Figure 3.1.

ments or exception throwing statements. Swami then backtracks from the visible-behavior statements to recursively fill in the variable value assignments according to the specification, resulting in a test template encoding the oracle, parameterized by test inputs. Swami then generates random, heuristic-driven test inputs to produce executable tests. Figure 3.3 shows automatically generated executable tests for the `Array(len)` constructor from the specification in Figure 3.1

Using natural language specifications pose numerous challenges. Consider the ECMA-262 specification of a JavaScript `Array(len)` constructor in Figure 3.1. The specification:



**Figure 3.4.** ECMA specifications include references to abstract operations, which are formally defined elsewhere in the specification document, but have no public interface. Section 9.6 of ECMA-262 (v5.1) specifies the abstract operation `ToUint32`, referenced in the specification in Figure 3.1.

- Uses natural language, such as “If the argument `len` is a Number and `ToUint32(len)` is equal to `len`, then the `length` property of the newly constructed object is set to `ToUint32(len)`.”
- Refers to abstract operations defined elsewhere in the specification, such as `ToUint32`, which is defined in section 9.6 of the specification (Figure 3.4).
- Refers to implicit operations not formally defined by the specification, such as `min`, `max`, `is not equal to`, `is set to`, `is an element of`, and `is greater than`.
- Describes complex control flow, such as conditionals, using the outputs of abstract and implicit operations in other downstream operations and conditionals.

### 3.3 Evaluation

We evaluate Swami using ECMA-262, the official specification of the JavaScript programming language [188], and two well known JavaScript implementations: Java Rhino and C++ Node.js built on Chrome’s V8 JavaScript engine. We find that:

- Of the tests Swami generates, 60.3% are innocuous—they can never fail. Of the remaining tests, 98.4% are precise to the specification and only 1.6% are flawed and might raise false alarms.
- Swami generates tests that are complementary to developer-written tests. Our generated tests improved the coverage of the Rhino developer-written test suite and identified 1 previously unknown defect and 15 missing JavaScript features in Rhino, 1 previously unknown defect in Node.js, and 18 semantic ambiguities in the ECMA-262 specification.
- Swami also outperforms and complements state-of-the-art automated test generation techniques. Most tests generated by EvoSuite (which does not automatically extract oracles) that cover exceptional behavior are false alarms, whereas 98.4% of Swami-generated tests are correct tests that cannot result in false alarms. Augmenting EvoSuite-generated tests using Swami increased the statement coverage of 47 Rhino classes by, on average, 19.5%. Swami also produced fewer false alarms than Toradacu and Jdoctor, and, unlike those tools, generated tests for missing features.

While Swami’s regular-expression-based approach is rather rigid, it performs remarkably well in practice for exceptional and boundary behavior. It forms both a useful tool for generating tests for such behavior, and a baseline for further research into improving automated oracle extraction from natural language by using more advanced information retrieval and natural language processing techniques.

### 3.4 Comparison with the state-of-the-art test generation tools

Our research complements prior work on automatically generating test inputs for regression tests or manually-written oracles, such as EvoSuite [50] and Randoop [127], by automatically extracting oracles from natural language specifications. The closest work to ours is Toradacu [57] and Jdoctor [19], which focus on extracting oracles for exceptional be-

havior, and `@tComment` [169], which focuses on extracting preconditions related to nullness of parameters. These techniques are limited to using Javadoc comments, which are simpler than the specifications Swami tackles because Javadoc comments (1) provide specific annotations for pre- and post-conditions, including `@param`, `@throws`, and `@returns`, making them more formal [169]; (2) are collocated with the method implementations they specify, (3) use the variable names as they appear in the code, and (4) do not contain references to abstract operations specified elsewhere. Additionally, recent work showed that Javadoc comments are often out of date because developers forget to update them when requirements change [169]. Our work builds on `@tComment`, `Toradacu`, and `Jdoctor`, expanding the rule-based natural language processing techniques to apply to more complex and more natural language. Additionally, unlike those techniques, Swami can generate oracles for not only exceptional behavior but also boundary conditions. Finally, prior test generation work [19, 50, 57, 127] requires access to the source code to be tested, whereas Swami can generate black-box tests entirely from the specification document, without needing the source code.

### 3.5 Contributions

The main contributions of this work are:

- Swami, an approach for generating tests from structured natural language specifications.
- An open-source prototype Swami implementation, including rules for specification documents written in ECMA-script style, and the implementations of common abstract operations.
- An evaluation of Swami on the ECMA-262 JavaScript language specification, comparing Swami-generated tests to those written by developers and those automatically generated by `EvoSuite`, demonstrating that Swami generates tests often missed by de-

velopers and other tools and that lead to discovering several unknown defects in Rhino and Node.js.

- A replication package of all the artifacts and experiments described in paper available at <http://swami.cs.umass.edu/>.

The full version of this study [120] can be found at <https://doi.org/10.1109/ICSE.2019.00035>.

## CHAPTER 4

### IMPROVING FAULT LOCALIZATION

#### 4.1 Introduction

Identifying the defective program elements is the first step of repairing software defects whether manually or automatically. Fault localization research focuses on automatically identifying program elements (such as statements or methods) that are defective and cause software failures. Most of the automated fault localization techniques use dynamic analysis and run-time information of the buggy program to compute the suspiciousness score (probability of being defective) of the program elements. A ranked list of program elements can then be used by the developer or automated program repair technique to fix the defect. Please refer to the survey study [190] for more details.

Depending on the source of information used to localize the fault, fault localization (FL) techniques can be classified into multiple classes. For example, spectrum-based fault localization (SBFL) techniques (e.g., [3,65,192]) use test coverage information, mutation-based fault localization (MBFL) techniques (e.g., [119,128]) use test results collected from mutating the program, (dynamic) program slicing techniques (e.g., [6,141]) use the dynamic program dependencies, stack trace analysis techniques (e.g., [189,191]) use error messages, predicate switching techniques (e.g., [203]) use test results from mutating the results of conditional expressions, information retrieval-based fault localization (IR-based FL) techniques (e.g., [205], [151]) use bug report information, and history-based fault localization (e.g., [78,138]) use the development history to identify the suspicious program elements that are likely to be defective.

While there exists a variety of FL techniques, state-of-the-art automated program repair techniques typically use SBFL techniques because they are light weight and the test-suites are readily available for defects. However, recent studies have found that none of the class of FL techniques is the best and combining multiple techniques across different classes outperforms individual techniques [94, 207]. While CombineFL [207] combines 11 FL techniques which belong to seven different classes using support vector machines (SVM), DeepFL [94] combines 226 fault localization techniques which belong to four (SBFL, MBFL, Code complexity based, and IR-based FL) classes. Both these techniques are evaluated on the Defects4J dataset which is also popularly used for evaluating automated program repair techniques.

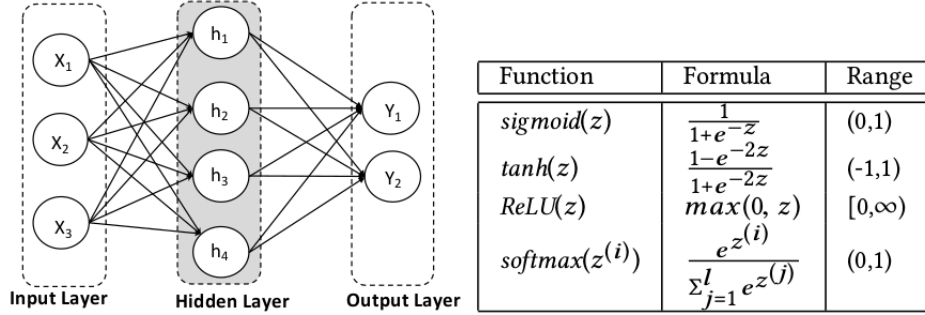
While researchers are actively working on improving fault localization, to the best of our knowledge, there does not exist any automated program repair technique that uses these advancements of combining multiple fault localization techniques that belong to different classes in the repair process. The work closest to experimenting with fault localization in program repair is a recent repair technique iFixR [80] which uses IR-based FL (instead of SBFL) and performs similar to using SBFL in terms of the number of fixed bugs plausibly/correctly. This is not surprising as either FL technique is likely to perform well. In this dissertation, we propose to use a combination of FL techniques in program repair process and hypothesize that with improved fault localization, the quality of repair techniques improves. This is work in progress.

## 4.2 Proposed Approach

### 4.2.1 Multi-layer Perceptron

Multi-layer Perceptron (MLP) is a basic class of feedforward Artificial Neural Networks (ANNs) which indicates that the network does not have any loop and the output of each node does not affect the node itself [133]. MLP is a supervised learning algorithm that learns a function  $f$  mapping from  $R^n$  to  $R^l$  by training a dataset which includes  $n$  features and  $l$  labels. It can learn a non-linear function approximator for either classification or regression





**Figure 4.1.** A Multi-Layer Perceptron with one input layer, one hidden layer and one output layer and the definitions for four widely used activation functions in neural networks.

problem. Assume that there is a set of training data  $D = (x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)$ , where  $x_i \in R^n$  and  $y_i \in R^l$ , and one hidden layer with  $k$  nodes, the function that MLP learns is as following:  $f(x) = \sigma_o(W_{ho}^T \sigma_h(W_{ih}^T x + b_h) + b_o)$  where  $W_{ih} \in R^{n \times k}$  represents the weights between input layer and hidden layer and  $W_{ho} \in R^{k \times l}$  represents the weights between hidden layer and output layer.  $b_h \in R^k$  and  $b_o \in R^l$  represent the bias of hidden layer and output layer, respectively.  $\sigma_h$  and  $\sigma_o$  represents the activation functions (such as  $\text{tanh}$ ,  $\text{ReLU}$ ,  $\text{sigmoid}$ , and  $\text{softmax}$ ) for the hidden layer and output layer, respectively.

Figure 4.1 shows a MLP with one input layer, one hidden layer and one output layer and the definitions for four widely used activation functions in neural networks ( $z^{(i)}$  represents the  $i$ th dimension of vector  $z$ ). Usually,  $\text{tanh}$ ,  $\text{ReLU}$ , and  $\text{sigmoid}$  are used as hidden layer activation function, while  $\text{softmax}$  is used as the output layer activation function for multi-class classification problems.

#### 4.2.2 Approach

Our approach to combine multiple FL techniques is inspired from DeepFL [94] and CombineFL [207] both of which first compute the suspiciousness scores for each program element using multiple FL techniques. The computed scores for each program element are represented as features of that element and the problem is framed as classifying the program element as *buggy* or *not buggy* based on its feature values. While DeepFL uses 225 such

features and uses neural networks, CombineFL uses 11 features and SVM, to compute the probability of a given program element to be buggy given its features.

Although, the source code and dataset of DeepFL is available as open source, it is not usable because (1) The DeepFL dataset does not provide mapping between the feature vectors created for program elements and the actual source code in Defects4J<sup>1</sup>, (2) using 226 fault localization techniques to compute features for each program element is not practical to be used in real-time repair, and (3) the features computed using IR-based FL techniques consider textual similarity between developer-written tests and source code as opposed to using natural-language bug reports. Similarly, the dataset and source code released for CombineFL does not consider IR-based FL and History-based techniques even though the results presented include both of these<sup>2</sup>.

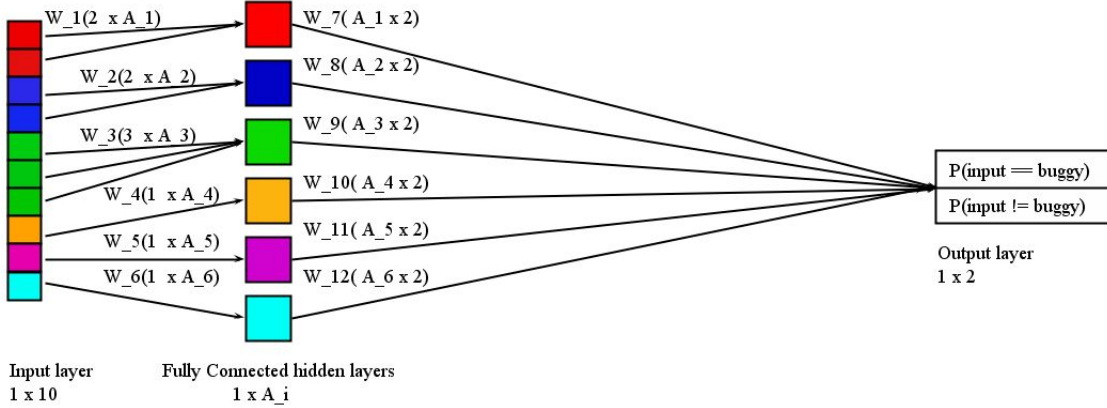
We therefore, propose to implement our own neural network based fault localization (NFL) technique that uses MLP to combine 10 different fault localization techniques which belong to six different classes: SBFL (ochiai,dstar); MBFL (metallaxis,muse); Dynamic program slicing (slicing union, slicing frequency, slicing intersection), Stack trace analysis, Predicate switching, and our own implementation of IR-based FL.

Figure 4.2 shows the high-level neural network architecture we propose to use. Different colors correspond to different class of fault localization techniques. The input to the model is a  $1 \times 10$  dimension *feature vector* that captures the suspiciousness scores of a program element (source code statement in this case) computed using 10 different fault localization techniques (we obtain IR-based FL features by implementing our own IR-based FL technique. Appendix A describes the details of our IR-based FL technique). The model first combines the scores of techniques that belong to the same family indicated by using different color coding for six hidden layers shown in figure 4.2). Next, the output of the hidden layers is combined together in a fully connected output layer of dimension  $1 \times 2$  which would contain

---

<sup>1</sup><https://github.com/DeepFL/DeepFaultLocalization/issues/2>

<sup>2</sup><https://damingz.github.io/combinefl/index.html>



**Figure 4.2.** The proposed MLP architecture to combine 10 different fault localization techniques which belong to six different classes. The input to the model is a  $1 \times 10$  dimension *feature vector* that captures the suspiciousness scores of a source code statement computed using 10 fault localization techniques. The output of the model is a  $1 \times 2$  dimension vector that contains the probabilities of the input to be buggy and not buggy.

the probabilities for the input to be buggy and not buggy. All the  $W_i$  are the network parameters which are learnt by the model during training.

While DeepFL uses a similar architecture, as the number of features in our case is significantly smaller (226 vs 10), the exact same architecture as of DeepFL may not work well for our dataset. Hence, we need to experiment and tune the proposed architecture by varying different hyperparameters as well as changing network architecture.

## 4.3 Evaluation

This section describes the datasets and evaluation metrics we will use to evaluate NFL.

### 4.3.1 Dataset

We will evaluate NFL on Defects4J defect benchmark which consists of 357 defects from five large open source Java projects and is used to evaluate several fault localization techniques including DeepFL and CombineFL.

We will also evaluate on Rhino dataset which consists of 13 defects mined from closed bug reports filed in the Rhino bug tracking system<sup>3</sup>.

### 4.3.2 Evaluation metrics

To evaluate NFL and compare its performance with the state-of-the-art FL techniques, we will use the following evaluation metrics used to evaluate CombineFL [207] in addition to the metrics described in Appendix A, Section A.6.

1. **E<sub>inspect</sub>@n**: It counts the number of successfully localized defects within the top n positions of the resultant ranked lists of program elements.
2. **EXAM**: It presents the percentage of program elements that have to be inspected until finding a faulty element.

---

<sup>3</sup><https://github.com/mozilla/rhino/issues>

## CHAPTER 5

# HIGH QUALITY REPAIR USING IMPROVED FAULT LOCALIZATION AND PATCH VALIDATION

### 5.1 Introduction

In this chapter we propose two methods to integrate the improved fault localization and patch validation in the automated program repair process along with the details about their evaluation. Both of these methods could improve the quality of repair techniques. Section 5.2 describes the first proposed method in which we propose to extend our Java Repair framework. Section 5.3 describes the second proposed method in which we use a neural network to do end-to-end repair. Both of these are work in progress.

### 5.2 Proposed Method 1: Extend Java Repair Framework to incorporate improved FL

Recall that we implement JaRFly, an open source Java Repair Framework for building Java Generate and Validate repair techniques which we use to evaluate the quality of repair techniques (Chapter 2, Section 2.3). The existing implementation of JaRFly decouples and provides high-level extension points for each of the fundamental components of the patch generation process (see Chapter 1, Section 1.2) which include specifying the problem representation, fitness function, mutation operators, and search strategy [64]. While JaRFly implements a variant of SBFL that uses configurable path weights to compute path-based localization, it also facilitates reading arbitrary localization data from a file, and an abstract class for implementing alternative fault localization strategies. We propose to extend JaRFly to incorporate our improved FL approach in the following manner. We can extend

JaRFly to incorporate IR-based FL described in Appendix A. Further, to integrate NFL, our proposed fault localization technique (Chapter 4) in JaRFly, we can use JaRFly’s feature to read localization data from a file. The extended JaRFly would allow us to re-evaluate the quality of the state-of-the-art repair techniques with proposed improvements in fault localization (Chapter 4) and patch validation test suites (Chapter 3).

### 5.2.1 Dataset

We will use following two datasets to evaluate this method.

1. **Defects4J**: defect benchmark which consists of 357 defects from five large open source Java projects and is popularly used to evaluate program repair techniques [74] . Figure 5.1 describes the Defects4J dataset.
2. **Rhino**: this consists of 30 defects which include 13 defects we manually mine from closed bug reports filed in the Rhino bug tracking system<sup>1</sup> and 17 defects which are used to evaluate Par [77].

identifier	project	description	KLoC	defects	tests	test KLoC
Chart	JFreeChart	Framework to create charts	85	26	222	42
Closure	Closure Compiler	JavaScript compiler	85	133	3,353	75
Lang	Apache Commons Lang	Extensions to the Java Lang API	19	65	173	31
Math	Apache Commons Math	Library of mathematical utilities	84	106	212	50
Time	Joda-Time	Date- and time-processing library	29	27	2,599	50
total			302	357	6,559	248

**Table 5.1.** The 357 defect dataset created from five real-world projects in the Defects4J version 1.1.0 benchmark. We used SLOccount to measure the lines of code (KLoC) counts (<https://www.dwheeler.com/sloccount/>). The **tests** and **test KLoC** columns refer to the developer-written tests.

---

<sup>1</sup><https://github.com/mozilla/rhino/issues>

### 5.2.2 Evaluation

To evaluate the extended JaRFly framework which implements three repair techniques, GenProg, Par, and TrpAutoRepair, we will perform following set of experiments.

1. **Baseline:** This involves running the techniques with default settings. We already have results for these experiments (Chapter 2, Section 2.3).
2. **Improved FL:** This involves using FL results computed using NFL (Chapter 4) to guide the repair process and then evaluate the quality of the repairs produced using held-out evaluation test suites.
3. **Improved Tests:** This involves using improved developer-written test suites by using Swami (Chapter 3) and other state-of-the-art test generation tools and then evaluating the quality of the repairs produced using held-out evaluation test suite.
4. **Improved FL and Tests:** This involves using fault localization results computed using NFL and improved developer-written test suites by using automated test generation tools to guide the repair process.

## 5.3 Proposed Method 2: Neural Network based End-to-End Program Repair

The state of the art neural machine translation techniques have become mature enough and outperform the traditional natural language processing in tasks such as translating from English to French. However, using such techniques for programming language is challenging because programs do not have a restricted vocabulary and the correctness criteria imposed by compilers are strict unlike humans who can understand the meaning of a natural language sentence even if a few words are misplaced or incorrect. Nevertheless, researchers have started exploring applying these techniques to the tasks such as code generation from program description [167] and have met some success. Researchers working on program repair have

also applied these techniques to automatically generate patches by *translating* buggy source code into repaired source code. However, the state-of-the-art deep-learning based program repair techniques can only fix syntax errors in small programs [83] or single-line defects in large programs [28] which requires manually specifying the defective source code statement. Further, the quality of the produced patches is quite low. All these are attributed to the limitation of deep learning models to manage the large vocabulary of programs and the large context of a defect to be fixed.

We propose to devise a novel neural network-based program repair (NPR) technique that can improve the quality of the patches of single-line defects produced by current tools and can also be used to patch complex multi-line real-world defects in large programs. We hypothesize that using convolution neural networks (CNNs) with multi-step attention [55] which is shown to outperform neural machine translation models that use recurrent neural networks (RNNs), along with copy mechanism [158], would enable our model to capture the large vocabulary and the defect context. We propose to implement this model which would take as input whole buggy file and produce a patch while considering the buggy context. The proposed model would use NFL for fault localization which combines multiple fault localization techniques (recall Chapter 4) and thus, would not require manually specifying defective program elements.

### 5.3.1 Problem Definition

We formally define the problem of automatically repairing a defective program as follows. Given a defective software program and a test suite which contains at least one failing test that exercises the defect in the program, we want to modify the software program such that the modified program passes all the tests. We next define the problem statement formally.

Given a buggy software system  $S^b$ , and a test-suite  $t$ , we first need to identify a set of source code lines  $l = \{l_1, l_2, \dots\}$  that are likely to have a bug. For each buggy line  $l_i$ , we would also know the buggy file  $f_{l_i}$ , the buggy class  $c_{l_i}$ , and the buggy method  $m_{l_i}$ . Thus,



for each buggy location  $l_i$ , we define a buggy context  $BC_i = \{f_{l_i}, c_{l_i}, m_{l_i}\}$ . For all  $l_i \in l$ , the problem is to predict a set of fixed lines  $f = \{f_1, f_2, \dots\}$  such that when we replace  $l_i$  with  $f_i$  incrementally to create a patched/modified software system  $S^f$  then  $S^f$  passes all the tests in test-suite  $t$ .

### 5.3.2 NPR Approach

We hypothesize that using convolution neural networks (CNN) with multi-step attention and copy mechanism would enable the model to capture the large vocabulary and the defect context enabling NPR to patch complex multi-line defects in large programs. For more details on advantages of using CNNs over RNNs see [55]. We propose to implement a bidirectional sequence-to-sequence neural machine translation model using CNNs with multi-step attention and copy mechanism. Figure 5.1 shows the high-level view of the training and testing of the proposed approach. There are four main components of the proposed architecture: an encoder, a decoder, an attention module, and a copy selector. During training, the model has access to both the buggy and the fixed source code lines. The model is trained to generate the best representations of the transformation from buggy to fixed lines. In practice, this is conducted by finding the best combination of weights ( $W_{random}$  in Figure 5.1) that translates buggy lines in the training set to fixed lines. Multiple passes on the training data are required to obtain the best set of weights ( $W_{learned}$  in Figure 5.1). The copy selector learns to handle out of vocabulary tokens by copying rarely occurring tokens from buggy code to source code (e.g., specific variable names, method names etc.). During testing, since the model does not have access to the fixed line, the decoder processes tokens one by one, starting with a generic <START> token. The output of the decoder and the encoder are then combined through the multi-step attention module and the copy selector module. Finally, new tokens are generated based on the output of the copy selector, the attention, the encoder and the decoder. The generated token is then fed back to the decoder until the <END> token is generated.

Formally, the proposed approach will take as input the buggy software system  $S^b$ , and a test-suite  $t$  which contains passing and failing tests. It will use NFL to identify set of source code lines  $l = \{l_1, l_2, \dots\}$  that are likely to have a bug. Next, for each buggy line, the approach determines the buggy context  $BC_i = \{f_i, c_i, m_i\}$  and if the lines are contiguous (in the same file) then the proposed model would predict the modifications to be made to each line incrementally until the prediction stops by generating  $\langle \text{END} \rangle$  token. The predicted modifications would then be applied to the source buggy program and validated against the test-suite  $t$ .

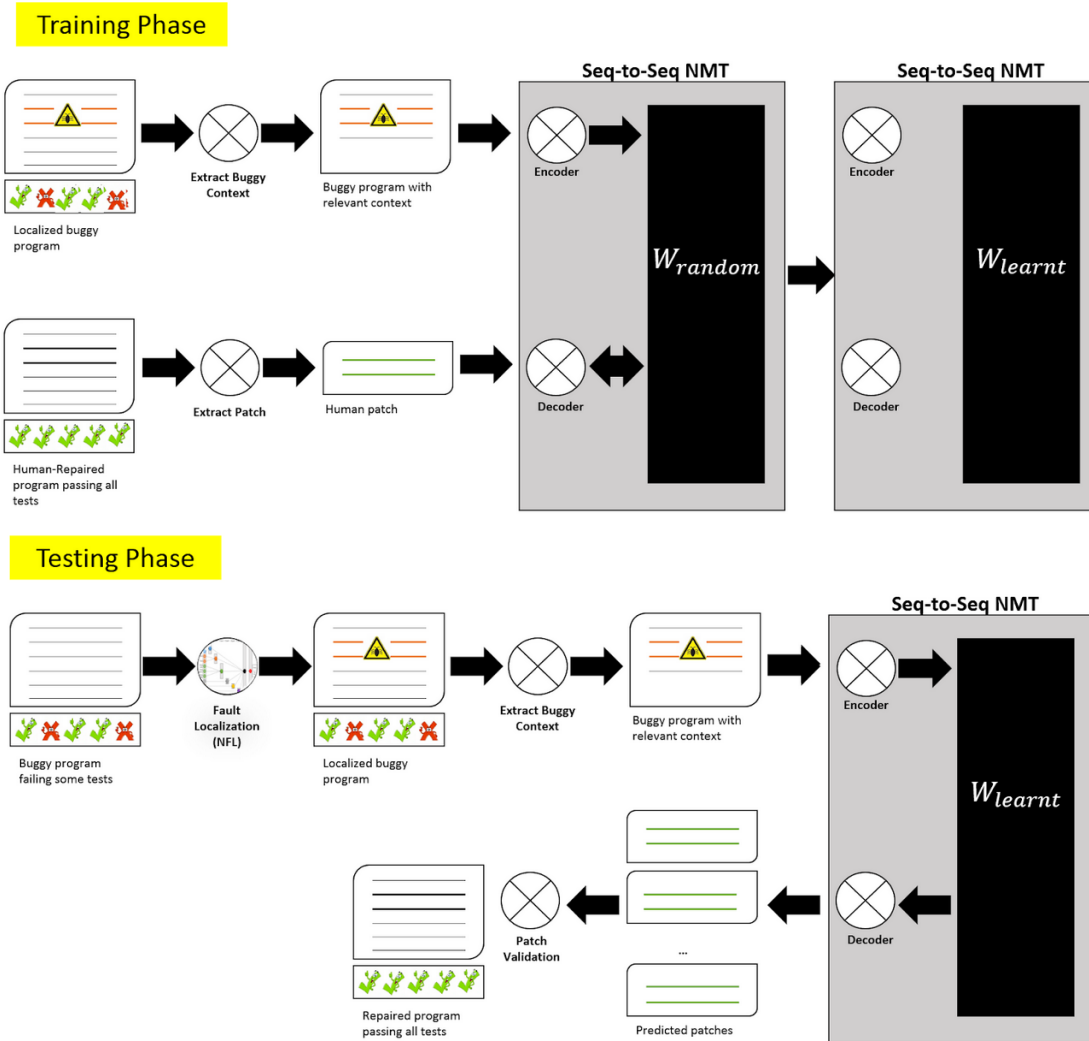
### 5.3.3 Dataset

For training our model, we will use the SequenceR [28] dataset which consists of pairs of buggy and patched code lines mined from multiple GitHub projects. The training dataset consists of 35,578 pairs and the test dataset consists of 4,711 pairs. To evaluate NPR on real work defects, we will use Defects4J [75] benchmark which is popularly used to evaluate program repair techniques.

### 5.3.4 Evaluation

We will evaluate NPR to identify for how many defects it is able to predict a plausible patch. The correctness of the predicted patches will then be determined using independently generated held-out evaluation test suites created using the human-fixed version of the software (recall Chapter 2, Section 2.3). The quality of a patch is then identified as the fraction of tests that pass in the evaluation test suite. Thus, a correct patch must pass all tests and therefore will have a 100% quality.

We shall describe the characteristics of defects which NPR can/cannot patch along with key factors that effect NPR's ability to produce correct patches. To summarize, we will report the accuracy of NPR to localize the defect and to predict the patch, its repair expressiveness, and repair quality. We will also report comparison with the existing state-of-the-art repair techniques.



**Figure 5.1.** The proposed end-to-end neural network-based program repair technique based on sequence-to-sequence neural machine translation (NMT) technique. During training, the model is fed pair of buggy source code and human-written patch. During testing, the model takes as input buggy source code and determines buggy program elements using NFL. Next, it uses the trained model to predict the candidate patches for the buggy program elements. Finally, the candidate patches are evaluated using provided test-suite to validate their correctness.

## CHAPTER 6

### RELATED WORK

This chapter describes the existing research work organized in the context of automated program repair (Section 6.1), studies of repair quality and other properties of automated program repair (Section 6.2), automated fault localization (Section 6.3), and automated test generation (Section 6.4).

#### 6.1 Automatic Program Repair

There are two classes of approaches to repairing defects using failing tests to identify faulty behavior and passing tests to encode desirable behavior: generate-and-validate and semantic-based repair. The generate-and-validate techniques use search-based software engineering [63] to generate many candidate patches and then validate them against tests. GenProg [90,92,185] uses a genetic programming heuristic [81] to search the space of candidate repairs. TrpAutoRepair [136] limits its patches to a single edit, uses random search instead of genetic programming, and heuristics to select which tests to run first, improving efficiency. Prophet [106] and HDRepair [88] automatically learn bug-fixing patterns from prior developer-written patches and use them to produce candidate patches for new defects. AE [183] is a deterministic technique that uses heuristic computation of program equivalence to prune the space of possible repairs, selectively choosing which tests to use to validate intermediate patch candidates. ErrDoc [172] uses insights obtained from a comprehensive study of error handling bugs in real-world C programs to automatically detect, diagnose, and repair the potential error handling bugs in C programs. JAID [27] uses automatically derived state abstractions from regular Java code without requiring any special annotations

and employs them, similar to the contract-based techniques to generate candidate repairs for Java programs. Qlose [36] optimizes a program distance, a function of syntactic and semantic differences between the original buggy and the patched programs, while generating candidate patches. DeepFix [62] and ELIXIR [153] use learned models to predict erroneous program locations along with patches. ssFix [193] uses existing code that is syntactically related to the context of a bug to produce patches. CapGen [187] works at the AST node level and uses context and dependency similarity (instead of semantic similarity) between the suspicious code fragment and the candidate code snippets to produce patches. SapFix [108] and Getafix [157], two tools deployed on production code at Facebook, efficiently produce correct repairs for large real-world programs. SapFix [108] uses prioritized repair strategies, including pre-defined fix templates, mutation operators, and bug-triggering change reverting, to produce repairs in realtime. Getafix [157] learns fix patterns from past code changes to suggest repairs for bugs that are found by Infer, Facebook’s in-house static analysis tool. SimFix [71] considers the variable name and method name similarity, as well as structural similarity between the suspicious code and candidate code snippets. Similar to CapGen, it prioritizes the candidate modifications by removing the ones that are found less frequently in existing patches. SketchFix [68] optimizes the candidate patch generation and evaluation by translating faulty programs to sketches (partial programs with holes) and lazily initializing the candidates of the sketches while validating them against the test execution. Par [77] and SOFix [98] use predefined repair templates to generate candidate patches. These repair templates are created based on the repair patterns mined from StackOverflow posts by comparing code samples in questions and answers for fine-grained modifications. Synthesis techniques can also construct new features from examples [30, 60], rather than address existing bugs.

The semantic-based techniques use semantic reasoning to synthesize patches to satisfy an inferred specification. Nopol [195], Semfix [124], DirectFix [111], and Angelix [112] use SMT or SAT constraints to encode test-based specifications. S3 [87] extends the semantics-

based family to incorporate a set of ranking criteria such as the variation of the execution traces similar to Qlose [36]. JFIX [86] extends Angelix [112] to target Java programs. SemGraft [110] infers specifications by symbolically analyzing a correct reference implementation instead of using test cases. Genesis [103], Refazer [147], NoFAQ [37], Sarfgen [178], and Clara [61] process correct patches to automatically infer code transformations to generate patches. SearchRepair [76] blurs the line between generate-and-validate and semantic-based techniques by using constraint-based encoding of the desired behavior to replace suspicious code with semantically-similar human-written code from elsewhere.

We propose methods to design a new repair technique as well as methods that can improve the quality of existing techniques. We also propose evaluation frameworks that aim to help researchers to properly evaluate their techniques' ability to produce high-quality patches for real-world defects. Our work enables properly comparing techniques with respect to patch quality, and encourages the creation of new techniques whose focus is producing high-quality patches on real-world defects. Empirical studies of fixes of real bugs in open-source projects can also improve repair techniques by helping designers select change operators and search strategies [74, 204]. Understanding how repair techniques handles particular classes of errors, such as security vulnerabilities [92, 132] can guide tool design. For this reason, some automated repair techniques focus on a particular defect class, such as buffer overruns [160, 162], unsafe integer use in C programs [31], single-variable atomicity violations [72], deadlock and livelock defects [96], concurrency errors [97], and data input errors [8] while other techniques tackle generic bugs. Although our evaluation focused on tools that fix generic bugs, our methodology can be applied to focused repair as well.

In addition to repair, search-based software engineering has been used for developing test suites [117, 176], finding safety violations [7], refactoring [159], and project management and effort estimation [11]. Good fitness functions are critical to search-based software engineering. Our findings indicate that using test cases alone as the fitness function leads to patches that

may not generalize to the program requirements, and more sophisticated fitness functions may be required for search-based program repair.

## 6.2 Empirical Studies Evaluating Automatic Program Repair

Prior work has argued the importance of evaluating the types of defects automated repair techniques can repair [121], and evaluating the generated patches for understandability, correctness, and completeness [118]. Yet many of the prior evaluations of repair techniques have focused on what fraction of a set of defects the technique can produce patches for (e.g., [25, 35, 42, 72, 92, 109, 183, 185]), how quickly they produce patches (e.g., [90, 183]), how maintainable the patches are (e.g., [52]), and how likely developers are to accept them (e.g., [2, 77]).

However, some recent studies have focused on evaluating the quality of repair and developing approaches to mitigate patch overfitting. For example, on 204 Eiffel defects, manual patch inspection showed that AutoFix produced high-quality patches for 51 (25%) of the defects, which corresponded to 59% of the patches it produced [131]. While AutoFix uses contracts to specify desired behavior, by contrast, the patch quality produced by techniques that use tests has been found to be much lower. Manual inspection of the patches produced by GenProg, TrpAutoRepair (referred to as RSRepair in that paper), and AE on a 105-defect subset of ManyBugs [137], and by GenProg, Nopol, and Kali on a 224-defect subset of Defects4J showed that patch quality is often lacking in automatically produced patches [109]. An automated evaluation approach that uses a second, independent test suite not used to produce the patch to evaluate the quality of the patch similarly showed that GenProg, TrpAutoRepair, and AE all produce patches that overfit to the supplied specification and fail to generalize to the intended specification [23, 163]. This work has led to new techniques that improve the quality of the patches [76, 104, 106, 193, 194, 200]. For example, DiffTGen generates tests that exercise behavior differences between the defective version and a candidate patch, and uses a human oracle to rule out incorrect patches. This

approach can filter out 49.4% of the overfitting patches [193]. Using heuristics to approximate oracles can generate more tests to filter out 56.3% of the overfitting patches [194]. UnsatGuided uses held-out tests to filter out overfitting patches for synthesis-based repair, and is effective for patches that introduce regressions but not for patches that only partially fix defects [200]. Automated test generation techniques that generate test inputs along with oracles [19, 57, 120, 169] or use behavioral domain constraints [10, 53], data constraints [48, 122, 123], or temporal constraints [12, 13, 15, 43, 126] as oracles could potentially address the limitations of the above-described approaches.

Using independent test suites to measure patch quality is imperfect, as test suites are partial and may identify some incorrect patches as correct. On a dataset of 189 patches produced by 8 repair techniques applied to 13 real-world Java projects, independent tests identify fewer than one fifth of the incorrect patches, underestimating the overfitting problem [85]. However, on other benchmarks, the results are much more positive. For example, on the QuixBugs benchmark, combining test-based and manual-inspection-based quality evaluation could identify 33 overfitting patches, while test-based evaluation alone identified 29 of the 33 (87.9%) [197]. While the human judgment is a criterion not used by the repair tools for patch construction, it is fundamentally different from the correctness criterion we use in our evaluation, as it is often difficult for humans to spot bugs even when told exactly where to look for them [129]. Further, using independently generated test suites instead of using the subset of the original test suite to evaluate patch quality ensures that we do not ignore regressions a patch is most likely to introduce. Poor-quality test suites result in patches that overfit to those suites [137]. Our evaluation goes further, demonstrating that high-quality, high-coverage test suites still lead to overfitting, and identifying other relationships between test suite properties and patch quality.

Our work has focused on understanding the effectiveness of repair techniques to patch large real-world Java programs correctly and to identify what factors affect the generation of high-quality patches. Studying the effects of test suite size, coverage, number of failing tests,



and test provenance on the quality of the patches generated by Angelix on the IntroClass [91] and Codeflaws [170] benchmarks of defects in small programs finds results consistent with ours. By contrast, our work focuses on real-world defects in real-world projects and generate-and-validate repair. Further, prior work has shown that the selection of test subjects (defects) can introduce evaluation bias [18, 135]. Our evaluation focuses precisely on the limits and potential of repair techniques on a large dataset of defects, and controls for a variety of potential confounds, addressing some of Monperrus’ concerns [118].

### 6.3 Automated Fault Localization

Fault Localization [3, 21, 34, 73, 95, 119, 130, 148, 149] aims to precisely identify potential buggy program elements that cause the defects to facilitate bug fixing. The most widely studied class of fault localization techniques is spectrum-based fault localization (SBFL) which usually apply statistical analysis (e.g., Tarantula [73], Ochiai [3], and Ample [34]) or learning techniques [21, 148, 149, 150] to the execution traces of both passed and failed tests to identify the most suspicious program elements (e.g., statements/methods). The insight behind these techniques is that program elements primarily executed by failed tests are more suspicious than the elements primarily executed by passed tests. However, a program element executed by a failed test does not necessarily indicate that the element has impact on the test execution and has caused the test failure. To bridge the gap between coverage and impact information, researchers proposed mutation-based fault localization (MBFL) [119, 128, 202], which injects changes to each program element (based on mutation testing [39, 70]) to check its impact on the test outcomes. MBFL has been applied to both general bugs (e.g., Metallaxis [128]) and regression bugs (e.g., FIFL [202]). Besides SBFL and MBFL, researchers have proposed to utilize various other information for fault localization such as program slicing (e.g., [6, 141]) that use dynamic program dependencies, stack trace analysis (e.g., [189, 191]) to use error messages, predicate switching (e.g., [203]) to use test results from mutating the results of conditional expressions, information retrieval-based

fault localization (IR-based FL) (e.g., [205], [152]) that use bug report information, and history-based fault localization (e.g., [78,138]) that use the development history to identify the suspicious program elements that are likely to be defective.

The work that is closest to our proposed approach of combining multiple fault localization techniques is DeepFL [94] and CombineFL [207] both of which first compute the suspiciousness scores for each program element using multiple FL techniques. Although, the source code and dataset of DeepFL is available as open source, it is not usable because (1) The DeepFL dataset does not provide mapping between the feature vectors created for program elements and the actual source code in Defects4J<sup>1</sup>, (2) using 226 fault localization techniques to compute features for each program element is not practical to be used in real-time repair, and (3) the features computed using IR-based FL techniques consider textual similarity between developer-written tests and source code as opposed to using natural-language bug reports. The dataset and source code released for CombineFL does not consider IR-based FL and History-based techniques even though the results presented include both of these<sup>2</sup>. Hence, we propose to implement our own model that addresses these limitations.

## 6.4 Automated Test Generation

Techniques that extract oracles from Javadoc specifications are the closest prior work to Swami, our proposed approach of automatically generating tests from natural language specifications. Toradacu [57] and Jdoctor [19] do this for exceptional behavior, and @tComment [169] for null pointers. These tools interface with EvoSuite or Randoop to reduce their false alarms. JDoctor, the latest such technique, combines pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications for pre-conditions, and normal and exceptional post-conditions. Our approach builds on these

---

<sup>1</sup><https://github.com/DeepFL/DeepFaultLocalization/issues/2>

<sup>2</sup><https://damingz.github.io/combinefl/index.html>

ideas but applies to more general specifications than Javadoc, with more complex natural language. Unlike these tools, Swami does not require access to the source code and generates tests only from the specification, while also handling boundary conditions. When structured specifications, Javadoc, and source code are all available, these techniques are likely complementary. Meanwhile, instead of generating tests, runtime verification of Java API specifications can discover bugs, but with high false-alarm rates [93].

Requirements tracing maps specifications, bug reports, and other artifacts to code elements [40, 66, 190], which is related to Swami’s *Section Identification* using the Okapi model [144, 165]. Static analysis techniques typically rely on similar information-retrieval-based approaches as Swami, e.g., BLUiR [152], for identifying code relevant to a bug report. Swami’s model is simpler, but works well in practice; recent studies have found it to outperform more complex models on both text and source code artifacts [139, 171].

Dynamic analysis can also aid tracing, e.g., in the way CERBERUS uses execution tracing and dependency pruning analysis [44]. Machine learning can aid tracing, e.g., via word embeddings to identify similarities between API documents, tutorials, and reference documents [199]. Unlike Swami, these approaches require large ground-truth training datasets. Future research will evaluate the impact of using more involved information retrieval models.

Automated test generation (e.g., EvoSuite [50] and Randoop [127]) and test fuzzing (e.g., afl [4]) generate test inputs. They require manually-specified oracles or oracles manually encoded in the code (e.g., assertions), or generate regression tests [50]. Swami’s oracles can complement these techniques. Differential testing can also produce oracles by comparing behavior of multiple implementations of the same specification [22, 26, 49, 155, 164, 196] (e.g., comparing the behavior of Node.js to that of Rhino), but requires multiple implementations, whereas Swami requires none.

Specification mining uses execution data to infer (typically) FSM-based specifications [1, 13, 14, 15, 56, 82, 84, 99, 100, 101, 102, 126, 140, 156]. TAUTOKO uses such specifications to generate tests, e.g., of sequences of method invocations on a data structure [33], then iteratively

improving the inferred model [33, 177]. These dynamic approaches rely on manually-written or simple oracles (e.g., the program should not crash) and are complementary to Swami, which uses natural language specifications to infer oracles. Work on generating tests for non-functional properties, such as software fairness, relies on oracles inferred by observing system behavior, e.g., by asserting that the behavior on inputs differing in a controlled way should be sufficiently similar [53], [24], [10]. Meanwhile, assertions on system data can also act as oracles [122, 123], and inferred causal relationships in data management systems [51, 113, 114] can help explain query results and suggest oracles for systems that rely on data management systems [116]. Such inference can also help debug errors [179, 180, 181] by tracking and using data provenance [115].

Dynamic invariant mining, e.g., Daikon [48], can infer oracles from test executions by observing arguments' values method return values [125]. Such oracles are a kind of regression testing, ensuring only that behavior does not change during software evolution. Korat uses formal specifications of pre- and post-conditions (e.g., written by the developer or inferred by invariant mining) to generate oracles and tests [20]. By contrast, Swami infers oracles from the specification and neither requires source code nor an existing test suite.

## CHAPTER 7

### RESEARCH PLAN

This chapter describes the research plan for all the projects and tasks that are part of this dissertation along with potential risks in projects which are work in progress.

#### 7.1 Projects and Timeline




Following is the list of projects and tasks that are part of this dissertation along with their progress.

1. **Auto-Repair Applicability:** Defining an objective evaluation framework to evaluate the applicability of automated program repair techniques and evaluate state-of-the-art repair techniques to determine if they are able to patch defects which are hard and important for developers. *This project is completed and published in EMSE '18 ([121]).*
2. **Java Repair Quality:** Large scale study to evaluate the quality of automated program repair techniques on real world defects and determine the factors that affect repair quality. *This project is completed and is under review in TSE.*
3. **SOSRepair:** Scaling SearchRepair, a semantics based repair technique that produces high quality patches for small programs to be applicable on real-world defects in large programs. *This project is completed and published in TSE '19 ([5]).*
4. **Swami:** Automatically generating precise tests with oracles from structured natural language software specifications. *This project is completed and published in ICSE '19 (Research track) [120].*

5. **IR-based FL:** Implementing Information Retrieval based fault localization technique to use bug reports for localizing defects. *This project is completed as part of the synthesis project.*
6. **NFL:** Implement Neural Network based fault localization that combines 10 different fault localization techniques which belong to six different classes of fault localization approach. *This project is work in progress.*
7. **JaRFly extension:** Extend JaRFly to incorporate IR-based FL and NFL and test if improving fault localization and developer-written tests improves the quality of existing repair techniques. *This project is work in progress.*
8. **NPR:** Implement Neural Network based end-to-end program repair which takes as input buggy program and associated artifacts, and produces a patched program. NPR uses NFL for fault localization. *This project is work in progress.*
9. **Dissertation:** Write dissertation and prepare presentation. *This is yet to begin.*

Figure 7.1 shows the detailed timeline of these tasks/projects. The research plan for the projects what are work in progress is as follows. We plan to implement NFL by the end of January 2020 and then extend JaRFly to incorporate NFL and IR-based FL and evaluate if the improved fault localization and developer-written test suites improve the quality of the repair techniques by the end of February 2020. In parallel, we will also work on NPR separately and if by FSE deadline in March we have any interesting results, we shall submit to FSE otherwise, based on the findings, we attempt to improvise results and perform more experiments to target ICSE in August 2020. We plan to finish the writing of dissertation and defending it by the end of August so as to graduate by September 2020.

Timeline	2015	2016	2017	2018	2019	2020
<b>Task/Project</b>						
Auto-Repair Applicability				EMSE'18		
Java Repair Quality						TSE'20
SOSRepair					TSE'19	
Swami					ICSE'19	
IR-based FL				Synthesis Project		
NFL						
JaRFly extension						
NPR						
Dissertation						

Timeline	Dec 2019	Jan 2020	Feb 2020	Mar 2020	Apr 2020	May 2020	June 2020	July 2020	Aug 2020	Sep 2020
<b>Task/Project</b>										
NFL										
JaRFly extension				FSE					ICSE	
NPR										
Dissertation										

**Figure 7.1.** Proposed timeline for this dissertation. Top chart shows the lists of all the projects that are pursued as part of this dissertation along with their timeline. The green color indicates the projects that are completed and the orange indicates the projects/tasks that are work in progress. Bottom chart shows the detailed timeline plan for completing the projects which are work in progress and the task of writing the dissertation.

## 7.2 Potential Risks and Contingency Plan

Following are the potential risks we may encounter in the projects which are work in progress and proposed ways to mitigate it.

1. **NFL does not outperform CombineFL or DeepFL.** In such case, we shall use CombineFL's approach by incorporating IR-based FL features.
2. **Unavailability of bug reports for all defects.** If this happens for a majority of the defects in our dataset then we can consider using IR-based FL on failing tests instead of bug reports (similar to DeepFL). If this happens for small subset of our dataset then we can set the IR-based FL feature value to be 0.5 (indicating equal probability of being buggy and not buggy) for all the source code elements in NFL.
3. **NPR doesn't scale to fix multi-line defects.** We can experiment with other state-of-the-art neural machine translation models and if nothing works then we shall at least have some insights about why these techniques are not able to scale.



## CHAPTER 8

### CONTRIBUTIONS

While existing automated program repair techniques can fix a large number of bugs in real-world software, most of the repairs produced are not correct or acceptable to the developers. This is a critical concern which prevents program repair techniques to be used in real-life software development processes. This dissertation proposes multiple methods to address this problem.

We define objective evaluation frameworks to evaluate the applicability and quality of the automated repair techniques and evaluate state-of-the-art repair techniques using our framework. We also evaluate the quality of SOSRepair, a novel semantics-based repair technique that builds upon SearchRepair, a semantics-based repair technique which produces high-quality patches for small programs, to run on real-world defects. The key findings from all these evaluations reveal that fault localization and test suites significantly affect the quality of the repair techniques.

This motivates us to propose a method to improve fault localization and test suites used to guide the repair by extracting information from multiple sources including natural-language software artifacts such as software specifications and bug reports. Finally, we propose two methods to improve the quality of repair techniques by integrating the improved fault localization and patch validation in the program repair process. The first method involves extending our Java Repair framework to incorporate improved FL and test suites. The second method involves using deep learning to design end-to-end program repair that uses improved fault localization and test suites.

## APPENDIX

### INFORMATION RETRIEVAL BASED FAULT LOCALIZATION

#### A.1 Abstract

Software specification is almost always expressed informally in natural language and free text. Examples include requirement specifications, design documents, manual pages, system development journals, error logs, and related maintenance reports. Identifying the parts of the source code that correspond to a specification is a prerequisite to tasks including program comprehension, maintenance, requirements tracing, and impact analysis. This process, commonly called concern location (or concept, or feature location) is one of the most common, important, and expensive activities undertaken by developers. In this work, we present IRFL — a method for automating this task by using static program analysis and a structural information retrieval model that uses code constructs, such as class, method, and variable names to identify the parts of the software that are relevant to a given concern. Our method works with the popular ECMAScript International standard for natural-language specifications, which Mozilla Rhino’s follows. We evaluate IRFL on Mozilla Rhino — a large, open-source implementation of JavaScript written entirely in Java. IRFL achieves the mean average precision (MAP) of 0.48 and the mean reciprocal rank (MRR) of 0.81.

#### A.2 Introduction

A *software concern* (also known as concept or feature in software engineering domain) is *any consideration that can impact the implementation of a program* [146]. Every line of code exists to satisfy some concern which may be described in many ways and at various levels of abstractions such as list of features from feature specification document, requirements from

software requirements specification document, design patterns and design elements from design document and, low level programming concerns such as code comments written to describe the implemented code.

Identifying parts of the source code that are related to a given concern is central to the software development, testing, and maintenance activities. The relationship between the concerns and the implemented source code is rarely documented [79]. This makes it difficult for the programmers to answer questions such as “Where are all the places that the `undo` feature is implemented?” (top-down analysis [175]) and “What is this piece of code for?” (bottom-up analysis [175]). Manually locating concerns in an implemented program can be very cumbersome and impractical especially for large programs where programmers may not be well versed with all the components of the program. Further, without proper understanding of the scattered nature of the concern implementation, programmers may make incorrect changes or neglect to make changes and test all the right places.

Several concern location techniques (e.g., concept assignment [17], feature location [41], requirements tracing [46], and bug localization [152,198,199,206]) aim to address this problem by automatically locating the program elements (e.g., classes, methods, variables) that are relevant to a given concern. While most of these techniques focus on software maintenance activities where the programmers are given a change request (e.g., a bug report) and they have to identify the relevant parts of code which should be updated to satisfy the change request, our focus is slightly different. We aim to devise a technique that can facilitate software testers to verify if their software satisfies given specifications. The first step toward achieving this goal is to retrieve all the parts of the source code that must comply to a given specification. Henceforth, we refer to specifications and concerns interchangeably.

In this work, we create IRFL to establish a baseline for locating concerns using static program analysis and a structured information-retrieval based (IR) model. We use IRFL to locate the relevant parts of the Rhino source code (Rhino version 1.5R6) for the concerns extracted from the ECMAScript standard specification document (*ECMA-262 v3*). IRFL

is built upon BLUiR [152]—a tool to facilitate fault localization and Indri [166]—an IR toolkit to perform information retrieval. We evaluate IRFL using the ground-truth data provided by Eaddy et al. [45] for the Rhino dataset.

The rest of this report is organized as follows. Section A.3 describes the related work, Section A.4 describes the Rhino dataset used in this study, Section A.5 provides the details of our concern location approach, Section A.6 describes the metrics we use to evaluate our approach, Section A.7 describes the experiments and results, Section A.8 describes the threats to validity and, Section A.9 describes conclusion and future work.

### A.3 Related Work

Several techniques have been proposed to trace software concerns to the source code, which employ multiple types of software and data analysis [41]. These techniques can be broadly classified into static, dynamic, and combined analysis based approaches.

Static-analysis based techniques use the source code and possible documentation available along with source code such as code comments for mapping concerns [134,152]. Based on the type of information used for tracing, these techniques are further classified into text-based techniques, which use text-based search; structural-based techniques, which use the structure of the documents while searching; and hybrid techniques which combine information from different sources. These techniques use information retrieval (IR) based mechanisms wherein the source code and related documentation are represented as documents and a concern is represented as a query. An IR based technique is then used to rank the source code documents based on the similarity score between the query and the source code documents. BLUiR (Bug Localization using IR) is a system that localizes the bugs in a source code by identifying the source code that is relevant for a given a bug report [152]. It uses a hybrid approach in which it uses TF-IDF — an IR-based technique to identify the similarity between the terms in bug reports and the source code and incorporates the source code structure by distinguishing the terms occurring in comments with names of classes, methods, and variables. It is built on

top of the IR toolkit Indri [166]. Poshyvanyk et al. [134] propose a concern location approach that uses Formal Concept Analysis (FCA) and IR. Their approach uses LSI—an advanced IR approach, to map textual descriptions of software features or bug reports to relevant parts of the source code and to generate a ranked list of source code documents. After generating this ranked list of source code documents, it selects the most relevant attributes (terms that are highly similar) from the best-ranked documents and clusters these documents using the FCA algorithm (borrowed from mathematical lattice theory). This results in a concept lattice that contains an annotated description for concept nodes along with the links to actual documents in the source code. The user can then browse the results by traversing the lattice and refine the query if required.

Combined analysis based techniques use the static analysis (as described above) along with dynamic analysis which involves executing a program and analyzing its runtime behavior to determine the program elements that get *activated* when the given natural language concern is *exercised*. CERBERUS [45] is a hybrid system for concern location that uses IR, execution tracing, and prune dependency analysis to map concerns to source code elements. For the static analysis, it improves on existing techniques by considering the structure of the source code. For the dynamic analysis, it improves on existing techniques by considering the field accesses as well as the method executions. In addition to these improvements, CERBERUS uses the prune dependency analysis to infer additional relevant program elements by analyzing different kinds of relationships between program elements.

Recently, researchers have started exploring machine-learning based approaches as opposed to traditional IR models. For instance, Ye et al. [199] propose a model based on word embeddings in which they train their model on API documents, tutorials, and reference documents to learn word embeddings and then aggregate them to estimate semantic similarities between documents. They evaluate their model empirically and find that the learned vector space embeddings lead to improvements in a previously explored fault localization and a newly defined task of linking API documents to computer programming questions.

## A.4 Dataset

The lack of well-established benchmarks and commonly accepted set of features associated with the source code that implements them which could be used to compare concern location techniques [41] limits us to select a somewhat ad-hoc dataset to evaluate our technique. We chose the Rhino dataset for our study because Eaddy et al. [45] constructed the ground-truth data by manually mapping the parts of the Rhino source code to the relevant concerns from the ECMA standard specification document (*ECMA-262-v3*). Sections 1 to 6 of the *ECMA-262-v3* document describe generic information such as definitions and notational conventions which are not directly applicable to Rhino’s implementation. Hence, we consider only those specifications for concern location (section 7 onwards) which are directly applicable to the implemented software. There are a total of 480 such concerns in the *ECMA-262 v3* and a total of 140 classes (Java source code files) in the Rhino version 1.5R6.

### A.4.1 Pre-processing

The *ECMA-262 v3* document contains 16 sections each of which contains multiple levels of sub-sections. We pre-process this document to extract the leaf-level sections which contain the description of specification and concatenate the titles of all the parent sections up to the root node to create a summary. Finally, we present each concern in the following XML format which has a concern id (section id concatenated with titles of all parent sections), summary (title of the section) and description (body of the section).

```
<specification>
<concern id=< ID >>
<summary> </summary>
<description> </description>
</concern>
.
.
```

<concern> ... </concern>

</specification>

Figure A.1 shows the XML representation of the specifications for Array.pop and Array.join operations extracted from *ECMA-262-v3*.

```
<concern id="15 - Native ECMAScript Objects/15.4 - Array Objects/15.4.4 - Properties of Array Prototype Object/15.4.4.5">
  <summary>join</summary>
  <description>The elements of the array are converted to strings, and these strings are then concatenated,
  separated by occurrences of the separator. If no separator is provided, a single comma is used as the separator. The join
  method takes one argument, separator, and performs the following steps: 1. Call the [[Get]] method of this object with
  argument &quot;length&quot;; 2. Call ToUint32(Result(1)). 3. If separator is undefined, let separator be the
  single-character string &quot;&quot;; 4. Call ToString(separator). 5. If Result(2) is zero, return the empty
  string. 6. Call the [[Get]] method of this object with argument &quot;0&quot;; 7. If Result(6) is undefined or null,
  use the empty string; otherwise, call ToString(Result(6)). 8. Let R be Result(7). 9. Let k be 1. 10. If k equals Result(2),
  return R. 11. Let S be a string value produced by concatenating R and Result(4). 12. Call the [[Get]] method of this
  object with argument ToString(k). 13. If Result(12) is undefined or null, use the empty string; otherwise, call
  ToString(Result(12)). 14. Let R be a string value produced by concatenating S and Result(13). 15. Increase k by 1. 16.
  Go to step 10. The length property of the join method is 1. NOTE The join function is intentionally generic; it does not
  require that its this value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method.
  Whether the join function can be applied successfully to a host object is implementation-dependent.
</description>
</concern>
<concern id="15 - Native ECMAScript Objects/15.4 - Array Objects/15.4.4 - Properties of Array Prototype Object/15.4.4.6">
  <summary>pop</summary>
  <description>The last element of the array is removed from the array and returned. 1. Call the [[Get]]
  method of this object with argument &quot;length&quot;; 2. Call ToUint32(Result(1)). 3. If Result(2) is not zero,
  go to step 6. 4. Call the [[Put]] method of this object with arguments &quot;length&quot; and Result(2).
  5. Return undefined. 6. Call ToString(Result(2)-1). 7. Call the [[Get]] method of this object with argument Result(6).
  8. Call the [[Delete]] method of this object with argument Result(6). 9. Call the [[Put]] method of this object with
  arguments &quot;length&quot; and (Result(2)-1). 10. Return Result(7). NOTE The pop function is intentionally generic;
  it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects
  for use as a method. Whether the pop function can be applied successfully to a host object is implementation-dependent.
</description>
</concern>
```

**Figure A.1.** Specifications for Array join and Array pop operations from *ECMA-262-v3* represented in XML format

Analyzing the ground-truth data we found that the mappings created between concerns and source code were at a more granular level (method level instead of class level). We processed the ground-truth data to create the mappings at a class level which is required for evaluating our technique. After processing the ground-truth data, we got 279 out of the 480 concerns mapped to 90 out of 140 classes resulting in a total of 18,501 mappings. We evaluate the performance of our technique using these 279 concerns for which the ground truth data is available.

#### 15.4.4.5 Array.prototype.join (separator)

The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the *separator*. If no separator is provided, a single comma is used as the separator.

The `join` method takes one argument, *separator*, and performs the following steps:

1. Call the `[[Get]]` method of this object with argument `"length"`.
2. Call `ToInt32(Result(1))`.
3. If *separator* is **undefined**, let *separator* be the single-character string `" , "`.
4. Call `ToString(separator)`.
5. If `Result(2)` is zero, return the empty string.
6. Call the `[[Get]]` method of this object with argument `"0"`.
7. If `Result(6)` is **undefined** or **null**, use the empty string; otherwise, call `ToString(Result(6))`.
8. Let *R* be `Result(7)`.
9. Let *k* be 1.
10. If *k* equals `Result(2)`, return *R*.
11. Let *S* be a string value produced by concatenating *R* and `Result(4)`.
12. Call the `[[Get]]` method of this object with argument `ToString(k)`.
13. If `Result(12)` is **undefined** or **null**, use the empty string; otherwise, call `ToString(Result(12))`.
14. Let *R* be a string value produced by concatenating *S* and `Result(13)`.
15. Increase *k* by 1.
16. Go to step 10.

The `length` property of the `join` method is 1.

#### NOTE

The `join` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the `join` function can be applied successfully to a host object is implementation-dependent.

**Figure A.2.** Natural language concern from *ECMA-262-v3* describing the implementation of `Array.join` operation in Rhino

```
private static String js_join(Context cx, Scriptable thisObj,
                             Object[] args)
{
    String separator;

    long llength = getLengthProperty(thisObj);
    int length = (int)llength;
    if (llength != length) {
        throw Context.reportRuntimeError1(
            "msg.arraylength.too.big", String.valueOf(llength));
    }
    // if no args, use ", " as separator
    if (args.length < 1 || args[0] == Undefined.instance) {
        separator = ", ";
    } else {
        separator = ScriptRuntime.toString(args[0]);
    }
    if (length == 0) {
        return "";
    }
    String[] buf = new String[length];
    int total_size = 0;
    for (int i = 0; i != length; i++) {
        Object temp = getElement(thisObj, i);
        if (temp != null && temp != Undefined.instance) {
            String str = ScriptRuntime.toString(temp);
            total_size += str.length();
            buf[i] = str;
        }
    }
    total_size += (length - 1) * separator.length();
    StringBuffer sb = new StringBuffer(total_size);
    for (int i = 0; i != length; i++) {
        if (i != 0) {
            sb.append(separator);
        }
        String str = buf[i];
        if (str != null) {
            // str == null for undefined or null
            sb.append(str);
        }
    }
    return sb.toString();
}
```

**Figure A.3.** `Array.join` operation implemented in `NativeArray.java` file of Rhino Version 1.5R6 source code



## A.5 Approach

### A.5.1 Background

The fundamental assumption underlying IR-based concern location techniques is that some terms in a given concern will be found in relevant source files. Figure A.2 shows an example of natural language concern from *ECMA-262-v3* that describes the implementation of the `Array.join` operation in Rhino and Figure A.3 shows the actual implementation in Rhino Version 1.5R6. It can be realized that these two do contain common terms.

In IR-based concern location techniques, source code files represent the document collection to search and each concern represents a search query. Finding candidate source code files that should satisfy the concern is then reduced to standard IR ranking of documents (source code files) based on estimated relevance to each query (concern). The better an IR system can interpret the concerns and source files, the more accurately it is expected to highly rank the source code files.

A typical IR system begins with the following three-step preprocessing: text normalization, stopword removal, and stemming. Normalization involves removing punctuation, performing case-folding, tokenizing terms, etc. to ultimately define the initial vocabulary of terms in which queries and documents will be represented. Next, a set of extraneous terms identified in a stopword list (e.g., to, the, be, etc.) are filtered out in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same underlying term (e.g., ran, running, run) to improve term matching between query and document.

While these three pre-processing steps are often given short shrift in describing IR approaches, they embody important trade offs that can significantly influence the ultimate success or failure of the retrieval model. For example, normalization can increase matches between query and document by case-folding (improving recall), but this can also introduce spurious matches as well (hurting precision). Similarly, while stopword removal can reduce unhelpful term matching (e.g., to), any stopword removed is almost certain to hurt matching

for some particular query (e.g., to be or not to be). Finally, stemming will increase recall by conflating variants of the same underlying term, but this may also introduce false matches. For reproducible experimentation, preprocessing methods should be fully described along with other details of the IR model.

Once queries and documents have been pre-processed, documents are *indexed* by collecting and storing various statistics, such as *term frequency* (TF, the number of times a term occurs in a given document), and *document frequency* (DF, the number of documents in which the term appears). IDF refers to inverse document frequency, which is most simply formulated as  $\log(\frac{DF}{N})$  where N is the number of documents in the collection.

A widespread misconception about TF-IDF merits particular attention. Specifically, “The TF-IDF model is often used as a baseline model for comparison with new retrieval models. However, it is not actually a well-defined model, in the sense that there are several heuristic components in the model that can affect performance significantly.” [201].

### A.5.2 Architecture

We implement IRFL , which builds upon BLUiR [152] and use it to locate concerns in the source code. Figure A.4 shows the overall architecture of IRFL . The following sub-sections describe the working of IRFL in detail.

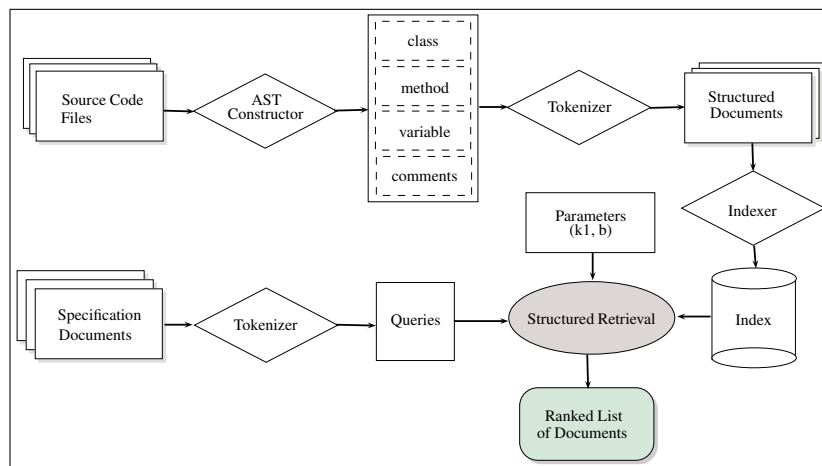


Figure A.4. IRFL Architecture

### A.5.2.1 Source Code Parsing & Term Indexing

First, IRFL takes as input the source code files (Java files of Rhino Version 1.5R6) in which we would like to locate concerns. Next, it builds the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT) and traverses the AST to extract different program constructs namely class names, method names, variable names, and comments. This ensures that the language keywords such as `string`, `class`, `if`, and `else` are not pruned off.

The extracted identifiers are then split into tokens using *CamelCase* splitting for indexing which improves the recall. As a specification document also contains full identifiers, the model indexes both full identifiers as well as split tokens. This information for each source code file is then stored as a structured XML document.

Reducing concern location to a standard IR task enables us to use prior theoretical and empirical IR methodology for tackling concern location. We adopt the Indri toolkit [166] for efficient indexing and developing our retrieval model. After XML documents are created above, they are handed off to Indri for stopword removal, stemming, and indexing.

### A.5.2.2 Specification Parsing & Query Generation

As described in Section A.4 (Pre-processing), IRFL parses the *ECMA-262 v3* document to extract all the specifications applicable to Rhino’s implementation and represents them in XML format (labeled as Queries in Figure A.4). Similar to source code files, the model tokenizes the XML-based concerns and then hands off to Indri for stopword removal, stemming and retrieval.

### A.5.2.3 Retrieval Model

This section describes the general Term Frequency — Inverse Document Frequency (TF-IDF) model and its variant (built in Indri [201]) which we adopt in our retrieval model.

Assume that a document is represented by a term frequency vector  $\vec{d}$  and a query by a term frequency vector  $\vec{q}$ . Both  $\vec{d}$  and  $\vec{q}$  are of length  $n$  where  $n$  is the size of vocabulary.

$$\vec{d} = (tf_d(t_1), tf_d(t_2), \dots, tf_d(t_n))$$

$$\vec{q} = (tf_q(t_1), tf_q(t_2), \dots, tf_q(t_n))$$

where  $tf_d(t_i)$  and  $tf_q(t_i)$  denotes the frequency of occurrence of the  $i^{\text{th}}$  term of the vocabulary ( $t_i$ ) in document  $\vec{d}$  and query  $\vec{q}$  respectively.

Typically when documents are represented in a vector space model, instead of using the raw term frequencies, terms are weighted by a heuristic TF-IDF weighting formula. The Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Weighted vectors for  $\vec{d}$  and  $\vec{q}$  are thus:

$$\vec{d} = (tf_d(t_1).idf(t_1), tf_d(t_2).idf(t_2), \dots, tf_d(t_n).idf(t_n))$$

$$\vec{q} = (tf_q(t_1).idf(t_1), tf_q(t_2).idf(t_2), \dots, tf_q(t_n).idf(t_n))$$

In the simplest TF-IDF model, IDF of term  $t$  is computed using  $idf(t) = \log(N/N_t)$  where  $N$  denotes the total number of documents and  $N_t$  denotes the number of documents that contain term  $t$ . Now to identify the similarity between document  $d$  and query  $q$ , we compute the sum of  $tf_d(t).idf(t)$  scores for all the terms  $t$  that occur in query  $q$ .

As described earlier, the actual TF-IDF models that are used in practice differ greatly from the simplest model for improved accuracy [145, 161]. We adopt Indri's [201] TF-IDF model which is based on the well-established BM25 (Okapi) model [145]. The following describes this model in detail.

The formula for computing inverse document frequency of term  $t$  is smoothed as shown in Eq. A.1 to avoid division by zero which would occur whenever a particular term appears in all the documents.

$$idf(t) = \log\left(\frac{N + 1}{N_t + 0.5}\right) \tag{A.1}$$

where  $N$  denotes the total number of documents and  $N_t$  denotes the number of documents in which term  $t$  appears.

The formula for computing the term frequency of a term  $t$  in document  $\vec{d}$  is computed by the Okapi TF formula:

$$tf_d(t) = \frac{k_1 x}{x + k_1(1 - b + b \frac{l_d}{l_C})} \quad (\text{A.2})$$

where  $x$  denotes the number of times term  $t$  appears in document  $\vec{d}$ ,  $k_1$  and  $b$  are tuning parameters and  $l_d$  and  $l_C$  denote the document length and average document length respectively. The tuning parameter  $k_1 (>= 0)$  calibrates document term frequency scaling and parameter  $b \in [0, 1]$  is the document scaling factor which adds a heuristic of modeling document length. When the value of  $b$  is 1, the term weight is fully scaled by the document length and when the value of  $b$  is 0, no length normalization is applied.

The formula for query TF is defined similarly though  $b$  is set to 0 since the query is fixed across documents being compared, and thus normalization of query length is not needed.  $k_1$  is set to 1000 to obtain raw query term frequency because the probability of having the same term many times in a query is rare. This makes the query TF formula almost identical to the original BM25 query TF formula as shown in Equation A.3 where  $k_3 = k_1$ .

$$tf_q(t) = \frac{k_3 y}{y + k_3} \quad (\text{A.3})$$

where  $y$  is the number of times term  $t$  appears in query  $\vec{q}$ . Finally, the similarity score of document  $\vec{d}$  against query  $\vec{q}$  is computed using:

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^n tf_d(t_i) tf_q(t_i) idf(t_i)^2 \quad (\text{A.4})$$

**Incorporating Structural Information.** The TF-IDF model presented in Equation A.4 does not consider the program constructs i.e., each term in a source code file is considered having the same relevance with respect to the given query. Therefore, important information like class names and method names often get lost in the relatively large number

of variable names and comments terms due to the term weighting function (Equation A.2). Therefore, if a source code file with class name `Array` also contains 10 other variable names having the term `Array`, then the class name `Array` does not add much weight. Thus, if there is a concern related to class `Array`, it will rank another file higher if that file has the term `Array` more than 11 times even in the local variable names or comments.

The proposed model distinguishes different code constructs to overcome this problem. We distinguish two alternative query representations coming from different fields of the concern (the *summary* and the more verbose *description*). Parsing source code structure also lets us distinguish four different document fields: *class*, *method*, *variable*, *comments*. To exploit all of these different types of query and document representations, we perform a separate search for each of the eight (query representation, document field) combinations and then sum document scores across all eight searches. Equation A.5 shows how we compute the similarity scores of documents for a given query.

$$s'(\vec{d}, \vec{q}) = \sum_{r \in Q} \sum_{f \in D} s(d_f, q_r) \quad (\text{A.5})$$

where  $r$  is a particular query representation and  $f$  is a particular document field.

The benefit of this model is that terms appearing in multiple document fields are implicitly assigned higher weight, since the contribution from each term is summed over all fields in which it appears. While this method of integrating structural information is quite simple, more sophisticated methods for integrating structural information exist and could be explored in future work, e.g., doing a weighted combination rather than a simple sum, or better yet, weighting term frequencies rather than document fields to better control for term frequency saturation [143].

## A.6 Evaluation Metrics

We evaluate IRFL using the following metrics which are popularly used for evaluating IR systems.

1. *Success at Top N*: The number of concerns with at least one relevant source code file found in the top N ( $N \in \{1, 5, 10\}$ ) ranked results. This metric emphasizes early precision over total recall. As our evaluation is limited to the 279 concerns of the ground-truth data, the value of Success at Top N lies between 0 and 279.
2. *Mean Reciprocal Rank (MRR)*: The reciprocal rank of a query response is the multiplicative inverse of the rank of the first relevant document. Similar to Success at Top N, this metric emphasizes early precision over recall. MRR is the reciprocal rank averaged over all the queries:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}$$

where  $\text{rank}_i$  is the reciprocal rank for  $i^{\text{th}}$  query. The value of  $MRR$  lies between 0 and 1. Higher  $MRR$  indicates the model has higher precision.  $MRR = 1$  indicates that for every query, the most relevant document is ranked first.

3. *Mean Average Precision (MAP)*: The most commonly used metric for IR system evaluation. This metric takes into account all the documents retrieved for a query along with their ranks. Unlike the above metrics, this emphasizes on recall over precision.  $MAP$  for a set of queries is the mean of the average precision scores for each query:

$$\text{MAP} = \frac{\sum_{q=1}^Q \text{AP}(q)}{Q}$$

where  $\text{AP}(q)$  is the average precision of a single query  $q$ :

$$\text{AP} = \sum_{k=1}^M \frac{P(k) \times \text{pos}(k)}{\text{number of positive instances}}$$

where  $k$  is the rank of the document retrieved,  $M$  is the number of documents retrieved,  $\text{pos}(k)$  is the binary indicator of whether the document at  $k^{\text{th}}$  rank is relevant, and

$P(k)$  is the precision at the given cut-off rank  $k$ . The value of  $MAP$  lies between 0 and 1. Higher  $MAP$  indicates that model has higher recall.  $MAP = 1$  indicates that for every query, the model is able to retrieve all the relevant documents and assign them higher rank than non-relevant documents.

4. *Precision Per Query (PPQ)*: The fraction of relevant documents among the retrieved documents for a given concern:

$$PPQ_i = \frac{Relevant_i}{Retrieved_i}$$

where  $Relevant_i$  is the total number of relevant documents retrieved for the  $i^{\text{th}}$  concern and  $Retrieved_i$  is the total number of documents retrieved for the  $i^{\text{th}}$  concern. The value of  $PPQ$  lies between 0 and 1.

5. *Recall Per Query (RPQ)*: The fraction of relevant documents divided by the total number of relevant documents for a given concern.

$$RPQ_i = \frac{Relevant_i}{Total\ Relevant_i}$$

where  $Relevant_i$  is the total number of relevant documents retrieved for the  $i^{\text{th}}$  concern and  $Total\ Relevant_i$  is the total number of relevant documents that exists for the  $i^{\text{th}}$  concern. The value of  $RPQ$  lies between 0 and 1.

6. *Precision*: The model's precision is the average of the per-query precision, for all the queries:

$$Precision = \frac{1}{|Q|} \sum_{i=1}^{|Q|} PPQ_i$$

The value of *Precision* lies between 0 and 1.



7. *Recall*: The model’s recall is the average of the per-query recall, for all the queries:

$$\text{Recall} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} RPQ_i$$

The value of *Recall* lies between 0 and 1.

## A.7 Results

### A.7.1 Parameter Tuning

We tune the two parameters of our model—the term weight scaling parameter  $k_1$  and the document normalization parameter  $b$ . Traditional wisdom is to set  $k_1 = 1.2$  and  $b = 0.75$  however, since concern location is different from traditional text retrieval we did a linear sweep of all values between  $[0, 2]$  for  $k_1$  and  $[0, 1]$  for  $b$  with a step-size of 0.1 selecting  $k_1 = 0.5$  and  $b = 0.6$  as optimal. The performance of IRFL using these parameter values is slightly better than using default values. Figure A.5 shows the effect of tuning parameters on the performance of IRFL measured in terms of the evaluation metrics.

We experimented with two stemmers: Krovetz and Porter supported by Indri and did not observe any significant difference in the performance of IRFL. This is consistent with prior work [67, 152]) that shows that no single stemmer is better for all kinds of queries. Hence our model uses the Krovetz stemmer and the default stopword list provided with Indri.

Term Weighting	Top 1	Top 5	Top 10	MAP	MRR
$k_1 = 1000, b = 0$	157	260	276	0.44	0.73
$k_1 = 1.2, b = 0.75$	195	269	279	0.48	0.81
<b><math>k_1 = 0.5, b = 0.6</math></b>	<b>195</b>	<b>270</b>	<b>279</b>	<b>0.48</b>	<b>0.81</b>

**Figure A.5.** Performance of IRFL after tuning model parameters for the Rhino dataset. The optimal values of model parameters were found to be  $k_1 = 0.5$  and  $b = 0.6$ .

### A.7.2 Retrieval Results

IRFL is able to locate the source code files for all the 480 concerns extracted from the *ECMA-262-v3* document. However, the evaluation is limited to the 279 concerns for

which we have the correct mappings in the ground-truth dataset. Figure A.6 shows IRFL’s Success@TopN ( $N \in \{1, 5, 10, 10000\}$ ), MAP, MRR, Precision, and Recall for the 279 concerns of the ground-truth dataset. Note that 100% recall is impossible when limiting the number of retrieved documents by  $N$ . Considering all the documents retrieved (Top 10000), IRFL achieves a MAP score of 0.48 and MRR score of 0.81. The precision value of 45% indicates that, on average, 55% of the source code files retrieved are not relevant to the given concerns; however, the MRR score of 0.81 shows that relevant source code files are ranked higher, which is encouraging. The recall value of 89% indicates that on average, 89% of the relevant source code files are retrieved for given concerns however, the low MAP score of 0.48 indicates that all relevant source code files are not ranked higher than irrelevant ones. Considering top 10 ranked results, IRFL locates at least one relevant source code file for all 279 concerns correctly and considering top 5 ranked results, IRFL locates at least one relevant source code file for almost all (270) concerns. Considering top 1 ranked results show that for 195 out of 279 concerns, the relevant source code file is ranked first.

Top N	Success	MAP	MRR	Precision	Recall
Top 1	195	0.70	0.70	69.89	1.06
Top 5	270	0.73	0.80	54.05	4.09
Top 10	279	0.66	0.81	52.14	7.87
Top 10000	279	0.48	0.81	45.58	89.46

**Figure A.6.** IRFL’s performance considering top N ( $N \in \{1, 5, 10, 10000\}$ ) documents retrieved per concern.

Although incorporating document structure in the retrieval model (Equation A.5) adds some overhead to the retrieval process, total time taken to retrieve documents for all of the 480 concerns was  $\sim 10$  seconds which is reasonable.

### A.7.3 Failure Analysis

This section describes a qualitative analysis of the results described in section A.7.2. Figure A.7 shows the distribution of PPQ and RPQ computed for each of the 279 concerns of the ground-truth dataset considering all the retrieved results (top 10000). Note that

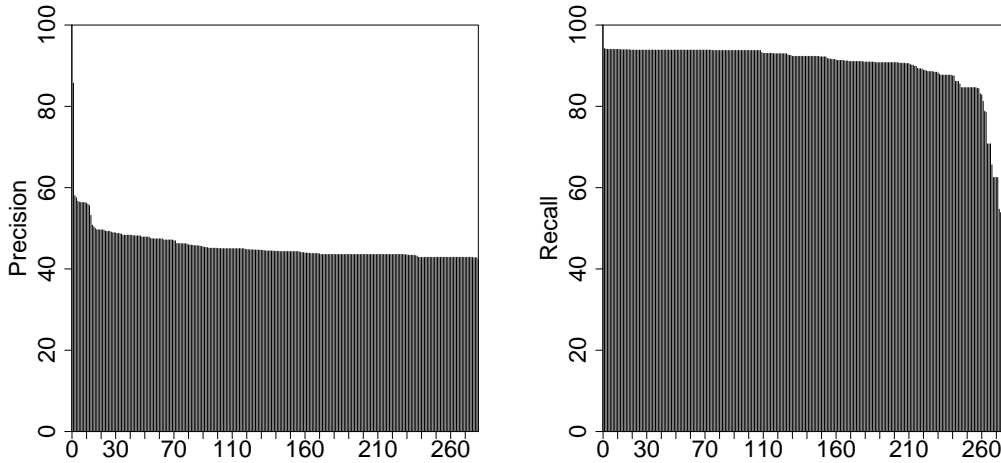
PPQ and RPQ indicate the performance of the technique without the rank of the retrieved documents. A PPQ score of  $\sim 45\text{--}50\%$  for a concern indicates that around 50% of all the documents retrieved for that concern are irrelevant documents however, a high MRR score of 0.81 indicates that relevant documents are ranked higher. The RPQ score, as expected, is high ( $> 85\%$ ) for most of the concerns; however, for certain concerns RPQ was found to be quite low (9%).

Investigating the concerns for which the relevant documents retrieved were too low, we found three types of scenarios where our proposed technique performs poorly.

The first type includes concerns for which IRFL couldn't retrieve any results are the ones that do not satisfy our assumption that common words exists between specifications and source code files. For instance, section 11.8 describes the implementation of relational operators e.g., a “Greater-than Operator” and the relevant source code e.g., Arguments.java which validates the arguments used to perform any allowable operations, doesn't contain any keywords which also occur in the specification of individual operators. Such scenarios can be handled using modern techniques that use semantics of the words.

The second type includes concerns that contain few words and refer to other concerns for more details. This often happens in all types of concern sources such as specification documents and bug reports. For instance section 15.3.5 of *ECMA-262-v3* document states that “*In addition to the required internal properties, every function instance has a `[[Call]]` property, a `[[Construct]]` property and a `[[Scope]]` property (see 8.6.2 and 13.2). The value of the `[[Class]]` property is “Function”.* where it refers to sections 8.6.2 and 13.2 that describe more details. A possible way to address such cases is to dereference the sections while preprocessing specification document to generate queries.

The third type includes the concerns that had very few words in their description. For instance, section 7.7 of *ECMA-262-v3* document describes the valid punctuators allowed in the JavaScript language and the query corresponding to it contains only four unique words



**Figure A.7.** Distribution of precision per query (PPQ) and recall per query (RPQ) considering all the documents retrieved for the 279 Rhino concerns of the ground-truth dataset.

“Punctuators”, “Syntax”, “Punctuator”, “one”, and “of”, and symbols { } ( ) [ ] . ; , > <, etc. It is hard to retrieve relevant source code files for such concerns.

## A.8 Threats to Validity

### A.8.1 Threats to internal validity:

The results presented in this study rely on the ground-truth data provided by Eaddy et al. [45], which involves manually annotating the Rhino source code with relevant concerns from *ECMA-262-v3*. Any human-error made in annotation will affect the results presented in this report.

### A.8.2 Threats to external validity:

The proposed method is generalizable to any kind of Java-based software provided the concerns from its specification can be represented in the form of generic XML-based template we use to represent concerns (recall Section A.4).

## A.9 Conclusion and Future Work

We propose IRFL , a structural information retrieval based concern location technique that uses program constructs, such as class and method names to accurately identify the source code which is relevant to a given concern. Our technique uses BLUiR for pre-processing and Indri—IR toolkit to perform the retrieval. We evaluate IRFL to identify the relevant source code files in Rhino version 1.5R6 for the *ECMA-262-v3* specifications. Considering the top 10 retrieved results for each specification, IRFL achieves the mean average precision (MAP) of 0.48 and the mean reciprocal rank (MRR) of 0.81.

In future, we plan to enhance IRFL by experimenting with more modern IR techniques such as LDA (e.g., [47]) and machine-learning based approaches that utilize the word-embeddings (e.g., [199]) to reduce the lexical gap between concerns and code which is usually identified as a significant impediment to the traditional IR-based approaches. We also intend to generalize our approach to incorporate the software written in other kinds of programming languages and concerns from other kinds of specification documents. Finally, we intend to create standard benchmarks and evaluation metrics which can be used for comparing different concern location techniques.

## BIBLIOGRAPHY

- [1] *Beyond support and confidence: Exploring interestingness measures for rule-based specification mining* (Montreal, ON, Canada, 2015).
- [2] Abd-El-Malek, Michael, Ganger, Gregory R., Goodson, Garth R., Reiter, Michael K., and Wylie, Jay J. Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles (SOSP)* (Brighton, UK, 2005), pp. 59–74.
- [3] Abreu, Rui, Zoetewij, Peter, and Gemund, Arjan JC Van. On the accuracy of spectrum-based fault localization. In *IEEE Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)* (Windsor, UK, Sept. 2007), pp. 89–98.
- [4] American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>, 2018.
- [5] Afzal, Afsoon, Motwani, Manish, Stolee, Kathryn, Brun, Yuriy, and Le Goues, Claire. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* (2019).
- [6] Agrawal, Hiralal, Horgan, Joseph R., London, Saul, and Wong, W. Eric. Fault localization using execution slices and dataflow tests. In *International Symposium on Software Reliability Engineering (ISSRE)* (1995), pp. 143–151.
- [7] Alba, Enrique, and Chicano, Francisco. Finding safety errors with ACO. In *Conference on Genetic and Evolutionary Computation (GECCO)* (London, England, UK, July 2007), pp. 1066–1073.
- [8] Alkhalaf, Muath, Aydin, Abdalbaki, and Bultan, Tevfik. Semantic differential repair for input validation and sanitization. In *International Symposium on Software Testing and Analysis (ISSTA)* (San Jose, CA, USA, July 2014), pp. 225–236.
- [9] Ammann, Paul, and Offutt, Jeff. *Introduction to Software Testing*, 1 ed. Cambridge University Press, New York, NY, USA, 2008.
- [10] Angell, Rico, Johnson, Brittany, Brun, Yuriy, and Meliou, Alexandra. Themis: Automatically testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) Demo track* (Lake Buena Vista, FL, USA, Nov. 2018), pp. 871–875.

- [11] Barreto, Ahilton, Barros, Márcio, and Werner, Cláudia. Staffing a software project: A constraint satisfaction approach. *Computers and Operations Research* 35, 10 (2008), 3073–3089.
- [12] Beschastnikh, Ivan, Brun, Yuriy, Abrahamson, Jenny, Ernst, Michael D., and Krishnamurthy, Arvind. Unifying FSM-inference algorithms through declarative specification. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, May 2013), pp. 252–261.
- [13] Beschastnikh, Ivan, Brun, Yuriy, Abrahamson, Jenny, Ernst, Michael D., and Krishnamurthy, Arvind. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Transactions on Software Engineering (TSE)* 41, 4 (April 2015), 408–428.
- [14] Beschastnikh, Ivan, Brun, Yuriy, Ernst, Michael D., and Krishnamurthy, Arvind. Inferring Models of Concurrent Systems from Logs of their Behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 468–479.
- [15] Beschastnikh, Ivan, Brun, Yuriy, Schneider, Sigurd, Sloan, Michael, and Ernst, Michael D. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Szeged, Hungary, September 2011), pp. 267–277.
- [16] Bhatia, Sahil, Kohli, Pushmeet, and Singh, Rishabh. Neuro-symbolic program corrector for introductory programming assignments. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 60–70.
- [17] Biggerstaff, Ted J, Mitbander, Bharat G, and Webster, Dallas. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering* (1993), IEEE Computer Society Press, pp. 482–498.
- [18] Bird, Christian, Bachmann, Adrian, Aune, Eirik, Duffy, John, Bernstein, Abraham, Filkov, Vladimir, and Devanbu, Premkumar. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, August 2009), pp. 121–130.
- [19] Blasi, Arianna, Goffi, Alberto, Kuznetsov, Konstantin, Gorla, Alessandra, Ernst, Michael D., Pezzè, Mauro, and Castellanos, Sergio Delgado. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, Netherlands, 2018), pp. 242–253.
- [20] Boyapati, Chandrasekhar, Khurshid, Sarfraz, and Marinov, Darko. Korat: Automated testing based on Java predicates. In *International Symposium on Software Testing and Analysis (ISSTA)* (Rome, Italy, 2002), pp. 123–133.

- [21] Briand, Lionel C, Labiche, Yvan, and Liu, Xuetao. Using machine learning to support debugging with tarantula. In *The 18th IEEE International Symposium on Software Reliability (ISSRE'07)* (2007), IEEE, pp. 137–146.
- [22] Brubaker, Chad, Jana, Suman, Ray, Baishakhi, Khurshid, Sarfraz, and Shmatikov, Vitaly. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (S&P)* (2014), pp. 114–129.
- [23] Brun, Yuriy, Barr, Earl, Xiao, Ming, Le Goues, Claire, and Devanbu, Prem. Evolution vs. intelligent design in program patching. Tech. Rep. <https://escholarship.org/uc/item/3z8926ks>, UC Davis: College of Engineering, 2013.
- [24] Brun, Yuriy, and Meliou, Alexandra. Software fairness. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results track* (Lake Buena Vista, FL, USA, Nov. 2018), pp. 754–759.
- [25] Carzaniga, Antonio, Gorla, Alessandra, Mattavelli, Andrea, Perino, Nicolò, and Pezzè, Mauro. Automatic recovery from runtime failures. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, 2013), pp. 782–791.
- [26] Chen, Jie, Xu, Xiwei, Osterweil, Leon J., Zhu, Liming, Brun, Yuriy, Bass, Len, Xiao, Junchao, Li, Mingshu, and Wang, Qing. Using Simulation to Evaluate Error Detection Strategies: A Case Study of Cloud-Based Deployment Processes. *Journal of Systems and Software* 110 (December 2015), 205–221.
- [27] Chen, Liushan, Pei, Yu, and Furia, Carlo A. Contract-based program repair without the contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana, IL, USA, Nov. 2017), pp. 637–647.
- [28] Chen, Zimin, Komrusch, Steve James, Tufano, Michele, Pouchet, Louis-Noël, Poshyvanyk, Denys, and Monperrus, Martin. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019).
- [29] Cheng, Xi, Zhou, Min, Song, Xiaoyu, Gu, Ming, and Sun, Jiaguang. IntPTI: Automatic integer error repair with proper-type inference. In *IEEE/ACM International Conference on Automated Software Engineering (ICSE)* (2017), pp. 996–1001.
- [30] Cochran, Robert, D’Antoni, Loris, Livshits, Benjamin, Molnar, David, and Veanes, Margus. Program boosting: Program synthesis via crowd-sourcing. In *Symposium on Principles of Programming Languages (POPL)* (Mumbai, India, January 2015), pp. 677–688.
- [31] Coker, Zack, and Hafiz, Munawar. Program transformations to fix C integers. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, 2013), pp. 792–801.



- [32] Cristian, Flaviu. Exception handling. Tech. Rep. RJ5724, IBM Research, 1987.
- [33] Dallmeier, Valentin, Knopp, Nikolai, Mallon, Christoph, Hack, Sebastian, and Zeller, Andreas. Generating test cases for specification mining. In *International Symposium on Software Testing and Analysis (ISSTA)* (Trento, Italy, 2010), pp. 85–96.
- [34] Dallmeier, Valentin, Lindig, Christian, and Zeller, Andreas. Lightweight defect localization for Java. In *European Conference on Object Oriented Programming (ECOOP)* (Glasgow, UK, 2005), pp. 528–550.
- [35] Dallmeier, Valentin, Zeller, Andreas, and Meyer, Bertrand. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering (ASE) short paper track* (Auckland, New Zealand, Nov. 2009), pp. 550–554.
- [36] D’Antoni, Loris, Samanta, Roopsha, and Singh, Rishabh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)* (Toronto, ON, Canada, July 2016), pp. 383–401.
- [37] D’Antoni, Loris, Singh, Rishabh, and Vaughn, Michael. NoFAQ: Synthesizing command repairs from examples. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ES-EC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 582–592.
- [38] DeMarco, Favio, Xuan, Jifeng, Berre, Daniel Le, and Monperrus, Martin. Automatic repair of buggy if conditions and missing preconditions with SMT. In *CSTVA* (2014), pp. 30–39.
- [39] DeMillo, Richard A, Lipton, Richard J, and Sayward, Frederick G. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (1978), 34–41.
- [40] Dit, Bogdan, Revelle, Meghan, Gethers, Malcom, and Poshyvanyk, Denys. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95.
- [41] Dit, Bogdan, Revelle, Meghan, Gethers, Malcom, and Poshyvanyk, Denys. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process* 25, 1 (2013), 53–95.
- [42] Durieux, Thomas, Martinez, Matias, Monperrus, Martin, Sommerard, Romain, and Xuan, Jifeng. Automatic repair of real bugs: An experience report on the Defects4J dataset. *CoRR abs/1505.07002* (2015).
- [43] Dwyer, Matthew B., Avrunin, George S., and Corbett, James C. Patterns in property specifications for finite-state verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (1999).

- [44] Eaddy, Marc, Aho, Alfred V., Antoniol, Giuliano, and Guéhéneuc, Yann-Gaël. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *IEEE International Conference on Program Comprehension (ICPC)* (Amsterdam, The Netherlands, 2008), pp. 53–62.
- [45] Eaddy, Marc, Aho, Alfred V., Antoniol, Giuliano, and Guéhéneuc, Yann-Gaël. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008.* (2008), IEEE, pp. 53–62.
- [46] Eaddy, Marc, Zimmermann, Thomas, Sherwood, Kaitlin D, Garg, Vibhav, Murphy, Gail C, Nagappan, Nachiappan, and Aho, Alfred V. Do crosscutting concerns cause defects? *IEEE transactions on Software Engineering* *34*, 4 (2008), 497–515.
- [47] Eddy, Brian P, Kraft, Nicholas A, and Gray, Jeff. Impact of structural weighting on a latent dirichlet allocation–based feature location technique. *Journal of Software: Evolution and Process* *30*, 1 (2018).
- [48] Ernst, Michael D., Cockrell, Jake, Griswold, William G., and Notkin, David. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)* *27*, 2 (2001), 99–123.
- [49] Evans, Robert B., and Savoia, Alberto. Differential testing: A new approach to change detection. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE) Poster track* (Dubrovnik, Croatia, 2007), pp. 549–552.
- [50] Fraser, Gordon, and Arcuri, Andrea. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)* *39*, 2 (February 2013), 276–291.
- [51] Freire, Cibele, Gatterbauer, Wolfgang, Immerman, Neil, and Meliou, Alexandra. A characterization of the complexity of resilience and responsibility for self-join-free conjunctive queries. *Proceedings of the VLDB Endowment (PVLDB)* *9*, 3 (2015), 180–191.
- [52] Fry, Zachary P., Landau, Bryan, and Weimer, Westley. A human study of patch maintainability. In *ISSTA* (2012), pp. 177–187.
- [53] Galhotra, Sainyam, Brun, Yuriy, and Meliou, Alexandra. Fairness testing: Testing software for discrimination. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, September 2017), pp. 498–510.
- [54] Gazzola, L., Micucci, D., and Mariani, L. Automatic software repair: A survey. *IEEE Transactions on Software Engineering (TSE)* *45*, 01 (Jan 2019), 34–67.
- [55] Gehring, Jonas, Auli, Michael, Grangier, David, Yarats, Denis, and Dauphin, Yann N. Convolutional sequence to sequence learning. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), JMLR. org, pp. 1243–1252.

- [56] Ghezzi, Carlo, Pezzè, Mauro, Sama, Michele, and Tamburrelli, Giordano. Mining behavior models from user-intensive web applications. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Hyderabad, India, 2014), pp. 277–287.
- [57] Goffi, Alberto, Gorla, Alessandra, Ernst, Michael D., and Pezzè, Mauro. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)* (Saarbrücken, Germany, July 2016), pp. 213–224.
- [58] Goodliffe, Pete. *Becoming a Better Programmer: A Handbook for People Who Care About Code.* ” O’Reilly Media, Inc.”, 2014.
- [59] Goues, Claire Le, Pradel, Michael, and Roychoudhury, Abhik. Automated program repair. *Commun. ACM* 62, 12 (Nov. 2019), 56–65.
- [60] Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. In *Symposium on Principles of Programming Languages (POPL)* (Austin, TX, USA, 2011), pp. 317–330.
- [61] Gulwani, Sumit, Radiček, Ivan, and Zuleger, Florian. Automated clustering and program repair for introductory programming assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA, June 2018), pp. 465–480.
- [62] Gupta, Rahul, Pal, Soham, Kanade, Aditya, and Shevade, Shirish K. DeepFix: Fixing common C language errors by deep learning. In *National Conference on Artificial Intelligence (AAAI)* (San Francisco, CA, USA, Feb. 2017), pp. 1345–1351.
- [63] Harman, Mark. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2007), pp. 342–357.
- [64] Harman, Mark, and Jones, Bryan F. Search-based software engineering. *Information and Software Technology* 43, 14 (2001), 833–839.
- [65] Harrold, Mary Jean, Rothermel, Gregg, Sayre, Kent, Wu, Rui, and Yi, Liu. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability* 10, 3 (2000), 171–194.
- [66] Hill, Emily, Rao, Shivani, and Kak, Avinash. On the use of stemming for concern location and bug localization in Java. In *SCAM* (2012), pp. 184–193.
- [67] Hill, Emily, Rao, Shivani, and Kak, Avinash. On the use of stemming for concern location and bug localization in java. In *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on* (2012), IEEE, pp. 184–193.
- [68] Hua, Jinru, Zhang, Mengshi, Wang, Kaiyuan, and Khurshid, Sarfraz. Towards practical program repair with on-demand candidate generation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, June 2018), pp. 12–23.

- [69] Jha, Susmit, Gulwani, Sumit, Seshia, Sanjit A., and Tiwari, Ashish. Oracle-guided component-based program synthesis. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Cape Town, South Africa, 2010), pp. 215–224.
- [70] Jia, Yue, and Harman, Mark. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.
- [71] Jiang, Jiajun, Xiong, Yingfei, Zhang, Hongyu, Gao, Qing, and Chen, Xiangqun. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Amsterdam, The Netherlands, July 2018), pp. 298–309.
- [72] Jin, Guoliang, Song, Linhai, Zhang, Wei, Lu, Shan, and Liblit, Ben. Automated atomicity-violation fixing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, CA, USA, 2011), pp. 389–400.
- [73] Jones, James A., Harrold, Mary Jean, and Stasko, John. Visualization of test information to assist fault localization. In *International Conference on Software Engineering (ICSE)* (Orlando, FL, USA, 2002), pp. 467–477.
- [74] Just, René, Jalali, Darioush, and Ernst, Michael D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA* (2014), pp. 437–440.
- [75] Just, René, Jalali, Darioush, and Ernst, Michael D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), ACM, pp. 437–440.
- [76] Ke, Yalin, Stolee, Kathryn T., Le Goues, Claire, and Brun, Yuriy. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE)* (Lincoln, NE, USA, November 2015), pp. 295–306.
- [77] Kim, Dongsun, Nam, Jaechang, Song, Jaewoo, and Kim, Sunghun. Automatic patch generation learned from human-written patches. In *ICSE* (2013), pp. 802–811.
- [78] Kim, Sunghun, Zimmermann, Thomas, Whitehead Jr, E James, and Zeller, Andreas. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society, pp. 489–498.
- [79] Ko, Andrew J, DeLine, Robert, and Venolia, Gina. Information needs in collocated software development teams. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on* (2007), IEEE, pp. 344–353.
- [80] Koyuncu, Anil, Liu, Kui, Bissyandé, Tegawendé F, Kim, Dongsun, Monperrus, Martin, Klein, Jacques, and Le Traon, Yves. ifixr: bug report driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2019), ACM, pp. 314–325.

- [81] Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
- [82] Krka, Ivo, Brun, Yuriy, Edwards, George, and Medvidovic, Nenad. Synthesizing partial component-level behavior models from system specifications. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, August 2009), pp. 305–314.
- [83] Kruthiventi, Srinivas SS, Ayush, Kumar, and Babu, R Venkatesh. Deepfix: A fully convolutional neural network for predicting human eye fixations. *IEEE Transactions on Image Processing* 26, 9 (2017), 4446–4456.
- [84] Le, Tien-Duy B., Le, Xuan Bach D., Lo, David, and Beschastnikh, Ivan. Synergizing specification miners through model fissions and fusions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Lincoln, NE, USA, November 2015).
- [85] Le, Xuan Bach D., Bao, Lingfeng, Lo, David, Xia, Xin, Li, Shanping, and Pasareanu, Corina S. On reliability of patch correctness assessment. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (May 2019).
- [86] Le, Xuan Bach D., Chu, Duc-Hiep, Lo, David, Le Goues, Claire, and Visser, Willem. JFIX: Semantics-based repair of Java programs via symbolic PathFinder. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Santa Barbara, CA, USA, July 2017), pp. 376–379.
- [87] Le, Xuan-Bach D., Chu, Duc-Hiep, Lo, David, Le Goues, Claire, and Visser, Willem. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, September 2017).
- [88] Le, Xuan Bach D., Lo, David, and Le Goues, Claire. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Mar. 2016), vol. 1, pp. 213–224.
- [89] Le, Xuan Bach D., Thung, Ferdian, Lo, David, and Goues, Claire Le. Overfitting in semantics-based automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 163–163.
- [90] Le Goues, Claire, Dewey-Vogt, Michael, Forrest, Stephanie, and Weimer, Westley. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, 2012), pp. 3–13.

- [91] Le Goues, Claire, Holtschulte, Neal, Smith, Edward K., Brun, Yuriy, Devanbu, Premkumar, Forrest, Stephanie, and Weimer, Westley. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE TSE* 41, 12 (December 2015), 1236–1256.
- [92] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)* 38 (2012), 54–72.
- [93] Legunsen, Owolabi, Hassan, Wajih Ul, Xu, Xinyue, Roşu, Grigore, and Marinov, Darko. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *ASE* (2016), pp. 602–613.
- [94] Li, Xia, Li, Wei, Zhang, Yuqun, and Zhang, Lingming. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2019), ACM, pp. 169–180.
- [95] Liblit, Ben, Naik, Mayur, Zheng, Alice X, Aiken, Alex, and Jordan, Michael I. Scalable statistical bug isolation. In *Acm Sigplan Notices* (2005), vol. 40, ACM, pp. 15–26.
- [96] Lin, Yiyan, and Kulkarni, Sandeep S. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In *International Symposium on Software Testing and Analysis (ISSTA)* (San Jose, CA, USA, July 2014), pp. 237–247.
- [97] Liu, Peng, Tripp, Omer, and Zhang, Charles. Grail: Context-aware fixing of concurrency bugs. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Hong Kong, China, Nov. 2014), pp. 318–329.
- [98] Liu, Xuliang, and Zhong, Hao. Mining StackOverflow for program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (Campobasso, Italy, Mar. 2018), pp. 118–129.
- [99] Lo, David, and Khoo, Siau-Cheng. QUARK: Empirical assessment of automaton-based specification miners. In *Working Conference on Reverse Engineering (WCRE)* (2006).
- [100] Lo, David, and Khoo, Siau-Cheng. SMAR TIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)* (Portland, OR, USA, 2006), pp. 265–275.
- [101] Lo, David, and Maoz, Shahar. Scenario-based and value-based specification mining: Better together. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Antwerp, Belgium, 2010), pp. 387–396.
- [102] Lo, David, Mariani, Leonardo, and Pezzè, Mauro. Automatic steering of behavioral model inference. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Amsterdam, The Netherlands, 2009), pp. 345–354.

- [103] Long, Fan, Amidon, Peter, and Rinard, Martin. Automatic inference of code transforms for patch generation. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 727–739.
- [104] Long, Fan, and Rinard, Martin. Staged program repair with condition synthesis. In *ESEC/FSE* (2015), pp. 166–178.
- [105] Long, Fan, and Rinard, Martin. An analysis of the search spaces for generate and validate patch generation systems. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Buenos Aires, Argentina, 2016), pp. 702–713.
- [106] Long, Fan, and Rinard, Martin. Automatic patch generation by learning correct code. In *POPL* (2016), pp. 298–312.
- [107] Macho, Christian, McIntosh, Shane, and Pinzger, Martin. Automatically repairing dependency-related build breakage. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2018), pp. 106–117.
- [108] Marginean, Alexandru, Bader, Johannes, Chandra, Satish, Harman, Mark, Jia, Yue, Mao, Ke, Mols, Alexander, and Scott, Andrew. SapFix: Automated end-to-end repair at scale. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Montreal, QC, Canada, May 2019).
- [109] Martinez, Matias, Durieux, Thomas, Sommerard, Romain, Xuan, Jifeng, and Monperus, Martin. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *EMSE* 22, 4 (April 2017), 1936–1964.
- [110] Mechtaev, Sergey, Nguyen, Manh-Dung, Noller, Yannic, Grunske, Lars, and Roychoudhury, Abhik. Semantic program repair using a reference implementation. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, 2018), pp. 129–139.
- [111] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. DirectFix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)* (Florence, Italy, May 2015).
- [112] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)* (Austin, TX, USA, May 2016).
- [113] Meliou, Alexandra, Gatterbauer, Wolfgang, Halpern, Joseph Y., Koch, Christoph, Moore, Katherine F., and Suciu, Dan. Causality in databases. *IEEE Data Engineering Bulletin* 33, 3 (2010), 59–67.
- [114] Meliou, Alexandra, Gatterbauer, Wolfgang, Moore, Katherine F., and Suciu, Dan. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment (PVLDB)* 4, 1 (2010), 34–45.

- [115] Meliou, Alexandra, Gatterbauer, Wolfgang, and Suciu, Dan. Bringing provenance to its full potential using causal reasoning. In *USENIX Workshop on the Theory and Practice of Provenance (TaPP)* (2011).
- [116] Meliou, Alexandra, Roy, Sudeepa, and Suciu, Dan. Causality and explanations in databases. *Proceedings of the VLDB Endowment (PVLDB) tutorial 7*, 13 (2014), 1715–1716.
- [117] Michael, Christoph C., McGraw, Gary, and Schatz, Michael A. Generating software test data by evolution. *IEEE Transactions on Software Engineering (TSE)* 27, 12 (Dec. 2001), 1085–1110.
- [118] Monperrus, Martin. A critical review of “Automatic patch generation learned from human-written patches”: Essay on the problem statement and the evaluation of automatic software repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 234–242.
- [119] Moon, Seokhyeon, Kim, Yunho, Kim, Moonzoo, and Yoo, Shin. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation* (2014), IEEE, pp. 153–162.
- [120] Motwani, Manish, and Brun, Yuriy. Automatically generating precise oracles from structured natural language specifications. In *Proceedings of the 41st International Conference on Software Engineering* (2019), IEEE Press, pp. 188–199.
- [121] Motwani, Manish, Sankaranarayanan, Sandhya, Just, René, and Brun, Yuriy. Do automated program repair techniques repair hard and important bugs? *EMSE* (2018).
- [122] Muşlu, Kıvanç, Brun, Yuriy, and Meliou, Alexandra. Data debugging with continuous testing. In *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) NIER track* (Saint Petersburg, Russia, August 2013), pp. 631–634.
- [123] Muşlu, Kıvanç, Brun, Yuriy, and Meliou, Alexandra. Preventing data errors with continuous testing. In *International Symposium on Software Testing and Analysis (ISSTA)* (Baltimore, MD, USA, July 2015), pp. 373–384.
- [124] Nguyen, Hoang Duong Thien, Qi, Dawei, Roychoudhury, Abhik, and Chandra, Satish. SemFix: Program repair via semantic analysis. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (San Francisco, CA, USA, 2013), pp. 772–781.
- [125] Nimmer, Jeremy W., and Ernst, Michael D. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis (ISSTA)* (Rome, Italy, July 2002).



- [126] Ohmann, Tony, Herzberg, Michael, Fiss, Sebastian, Halbert, Armand, Palyart, Marc, Beschastnikh, Ivan, and Brun, Yuriy. Behavioral Resource-Aware Model Inference. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Västerås, Sweden, September 2014), pp. 19–30.
- [127] Pacheco, Carlos, and Ernst, Michael D. Randoop: Feedback-directed random testing for Java. In *Conference on Object-oriented Programming Systems and Applications (OOPSLA)* (Montreal, QC, Canada, 2007), pp. 815–816.
- [128] Papadakis, Mike, and Le Traon, Yves. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.
- [129] Parnin, Chris, and Orso, Alessandro. Are automated debugging techniques actually helping programmers? In *International Symposium on Software Testing and Analysis (ISSTA)* (Toronto, ON, Canada, 2011), pp. 199–209.
- [130] Pearson, Spencer, Campos, José, Just, René, Fraser, Gordon, Abreu, Rui, Ernst, Michael D, Pang, Deric, and Keller, Benjamin. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering* (2017), IEEE Press, pp. 609–620.
- [131] Pei, Yu, Furia, Carlo A., Nordio, Martin, Wei, Yi, Meyer, Bertrand, and Zeller, Andreas. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering (TSE)* 40, 5 (2014), 427–449.
- [132] Perkins, Jeff H., Kim, Sunghun, Larsen, Sam, Amarasinghe, Saman, Bachrach, Jonathan, Carbin, Michael, Pacheco, Carlos, Sherwood, Frank, Sidiroglou, Stelios, Sullivan, Greg, Wong, Weng-Fai, Zibin, Yoav, Ernst, Michael D., and Rinard, Martin. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, October 12–14, 2009), pp. 87–102.
- [133] Popescu, Marius-Constantin, Balas, Valentina E, Perescu-Popescu, Liliana, and Mastorakis, Nikos. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems* 8, 7 (2009), 579–588.
- [134] Poshyvanyk, Denys, Gethers, Malcom, and Marcus, Andrian. Concept location using formal concept analysis and information retrieval. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 21, 4 (2012), 23.
- [135] Posnett, Daryl, Filkov, Vladimir, and Devanbu, Premkumar. Ecological inference in empirical software engineering. In *International Conference on Automated Software Engineering (ASE)* (Lawrence, KS, USA, November 2011), pp. 362–371.
- [136] Qi, Yuhua, Mao, Xiaoguang, and Lei, Yan. Efficient automated program repair through fault-recorded testing prioritization. In *ICSM* (Sept. 2013), pp. 180–189.
- [137] Qi, Zichao, Long, Fan, Achour, Sara, and Rinard, Martin. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA* (2015).

- [138] Rahman, Foyzur, Posnett, Daryl, Hindle, Abram, Barr, Earl, and Devanbu, Premkumar. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), ACM, pp. 322–331.
- [139] Rahman, Md Masudur, Chakraborty, Saikat, Kaiser, Gail, and Ray, Baishakhi. A case study on the impact of similarity measure on information retrieval based software engineering tasks. *CoRR abs/1808.02911* (2018).
- [140] Reiss, Steven P., and Renieris, Manos. Encoding program executions. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Toronto, ON, Canada, 2001), pp. 221–230.
- [141] Renieris, Manos, and Reiss, Steven P. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* (2003), IEEE, pp. 30–39.
- [142] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. NIST Planning Report 02-3, May 2002.
- [143] Robertson, Stephen, Zaragoza, Hugo, and Taylor, Michael. Simple bm25 extension to multiple weighted fields. In *Proceedings of the thirteenth ACM international conference on Information and knowledge management* (2004), ACM, pp. 42–49.
- [144] Robertson, Stephen E., Walker, Stephen, and Beaulieu, Micheline. Experimentation as a way of life: Okapi at TREC. *Information Processing and Management 36* (January 2000), 95–108.
- [145] Robertson, Stephen E., Walker, Steve, and Beaulieu, M. Experimentation as a way of life: Okapi at trec. *Information processing & management 36*, 1 (2000), 95–108.
- [146] Robillard, Martin P, and Murphy, Gail C. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM) 16*, 1 (2007), 3.
- [147] Rolim, Reudismam, Soares, Gustavo, D’Antoni, Loris, Polozov, Oleksandr, Gulwani, Sumit, Gheyi, Rohit, Suzuki, Ryo, and Hartmann, Björn. Learning syntactic program transformations from examples. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Buenos Aires, Argentina, May 2017), pp. 404–415.
- [148] Roychowdhury, Shounak, and Khurshid, Sarfraz. A novel framework for locating software faults using latent divergences. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2011), Springer, pp. 49–64.
- [149] Roychowdhury, Shounak, and Khurshid, Sarfraz. Software fault localization using feature selection. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering* (2011), ACM, pp. 11–18.

- [150] Roychowdhury, Shounak, and Khurshid, Sarfraz. A family of generalized entropies and its application to software fault localization. In *2012 6th IEEE International Conference Intelligent Systems* (2012), IEEE, pp. 368–373.
- [151] Saha, Ripon K., Lease, Matthew, Khurshid, Sarfraz, and Perry, Dewayne E. Improving bug localization using structured information retrieval. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Palo Alto, CA, USA, 2013), pp. 345–355.
- [152] Saha, Ripon K., Lease, Matthew, Khurshid, Sarfraz, and Perry, Dewayne E. Improving bug localization using structured information retrieval. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), IEEE, pp. 345–355.
- [153] Saha, Ripon K., Lyu, Yingjun, Yoshida, Hiroaki, and Prasad, Mukul R. ELIXIR: Effective object oriented program repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)* (Urbana, IL, USA, Nov. 2017), pp. 648–659.
- [154] Saha, Seemanta, Saha, Ripon K., and Prasad, Mukul R. Harnessing evolution for multi-hunk program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Montreal, QC, Canada, May 2019), pp. 13–24.
- [155] Samar, Vipin, and Patni, Sangeeta. Differential testing for variational analyses: Experience from developing KConfigReader. *CoRR abs/1706.09357* (2017).
- [156] Schur, Matthias, Roth, Andreas, and Zeller, Andreas. Mining behavior models from enterprise web applications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Saint Petersburg, Russia, 2013), pp. 422–432.
- [157] Scott, Andrew, Bader, Johannes, and Chandra, Satish. Getafix: Learning to fix bugs automatically. *CoRR abs/1902.06111* (2019).
- [158] See, Abigail, Liu, Peter J, and Manning, Christopher D. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).
- [159] Seng, Olaf, Stammel, Johannes, and Burkhart, David. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Conference on Genetic and Evolutionary Computation (GECCO)* (Seattle, WA, USA, July 2006), pp. 1909–1916.
- [160] Sidiroglou, Stelios, and Keromytis, Angelos D. Countering network worms through automatic patch generation. *IEEE Security and Privacy* 3, 6 (Nov. 2005), 41–49.
- [161] Singhal, Amit, et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.
- [162] Smirnov, Alexey, and cker Chiueh, Tzi. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium (NDSS)* (San Diego, CA, USA, Feb. 2005).

- [163] Smith, Edward K., Barr, Earl, Le Goues, Claire, and Brun, Yuriy. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE* (2015), pp. 532–543.
- [164] Srivastava, Varun, Bond, Michael D., McKinley, Kathryn S., and Shmatikov, Vitaly. A security policy oracle: Detecting security holes using multiple API implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, CA, USA, 2011), pp. 343–354.
- [165] Strohman, Trevor, Metzler, Donald, HowardTurtle, and Croft, W. Bruce. Indri: A language model-based search engine for complex queries. In *International Conference on Intelligence Analysis* (2005), pp. 2–6.
- [166] Strohman, Trevor, Metzler, Donald, Turtle, Howard, and Croft, W Bruce. Indri: A language model-based search engine for complex queries. In *International Conference on Intelligent Analysis* (2005), vol. 2, Amherst, MA, USA, pp. 2–6.
- [167] Sun, Zeyu, Zhu, Qihao, Mou, Lili, Xiong, Yingfei, Li, Ge, and Zhang, Lu. A grammar-based structural cnn decoder for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 7055–7062.
- [168] Tan, Shin Hwei, Dong, Zhen, Gao, Xiang, and Roychoudhury, Abhik. Repairing crashes in android apps. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 187–198.
- [169] Tan, Shin Hwei, Marinov, Darko, Tan, Lin, and Leavens, Gary T. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation (ICST)* (Montreal, QC, Canada, 2012), pp. 260–269.
- [170] Tan, Shin Hwei, Yi, Jooyong, Mechtaev, Sergey, and Roychoudhury, Abhik. Code-flaws: A programming competition benchmark for evaluating automated program repair tools. In *IEEE International Conference on Software Engineering Poster Track* (Buenos Aires, Argentina, May 2017), pp. 180–182.
- [171] Tantithamthavorn, Chakkrit, Lemma, Surafel Abebe, Hassan, Ahmed E., Ihara, Akinori, and Matsumoto, Kenichi. The impact of IR-based classifier configuration on the performance and the effort of method-level bug localization. *Information and Software Technology* (2018).
- [172] Tian, Yuchi, and Ray, Baishakhi. Automatically diagnosing and repairing error handling bugs in C. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)* (Paderborn, Germany, Sept. 2017), pp. 752–762.
- [173] van Tonder, Rijnard, and Goues, Claire Le. Static automated program repair for heap properties. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2018), pp. 151–162.

- [174] von Essen, Christian, and Jobstmann, Barbara. Program repair without regret. *Formal Methods in System Design* 47, 1 (2015), 26–50.
- [175] von Mayrhauser, Anneliese, Vans, A Marie, and Howe, Adele E. Program understanding behaviour during enhancement of large-scale software. *Journal of Software: Evolution and Process* 9, 5 (1997), 299–327.
- [176] Walcott, Kristen R., Soffa, Mary Lou, Kapfhammer, Gregory M., and Roos, Robert S. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)* (Portland, ME, USA, July 2006), pp. 1–12.
- [177] Walls, Robert J., Brun, Yuriy, Liberatore, Marc, and Levine, Brian Neil. Discovering Specification Violations in Networked Software Systems. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)* (Gaithersburg, MD, USA, November 2015), pp. 496–506.
- [178] Wang, Ke, Singh, Rishabh, and Su, Zhendong. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA, June 2018), pp. 481–495.
- [179] Wang, Xiaolan, Dong, Xin Luna, and Meliou, Alexandra. Data X-Ray: A diagnostic tool for data errors. In *International Conference on Management of Data (SIGMOD)* (2015).
- [180] Wang, Xiaolan, Meliou, Alexandra, and Wu, Eugene. QFix: Demonstrating error diagnosis in query histories. In *SIGMOD Demo* (2016), pp. 2177–2180.
- [181] Wang, Xiaolan, Meliou, Alexandra, and Wu, Eugene. QFix: Diagnosing errors through query histories. In *SIGMOD* (2017), pp. 1369–1384.
- [182] Wei, Yi, Pei, Yu, Furia, Carlo A., Silva, Lucas S., Buchholz, Stefan, Meyer, Bertrand, and Zeller, Andreas. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis (ISSTA)* (Trento, Italy, 2010), pp. 61–72.
- [183] Weimer, Westley, Fry, Zachary P., and Forrest, Stephanie. Leveraging program equivalence for adaptive program repair: Models and first results. In *ASE* (2013).
- [184] Weimer, Westley, and Necula, George C. Finding and preventing run-time error handling mistakes. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (Vancouver, BC, Canada, 2004), pp. 419–431.
- [185] Weimer, Westley, Nguyen, ThanhVu, Le Goues, Claire, and Forrest, Stephanie. Automatically finding patches using genetic programming. In *ICSE* (2009), pp. 364–374.
- [186] Weiss, Cathrin, Premraj, Rahul, Zimmermann, Thomas, and Zeller, Andreas. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007)* (2007), IEEE, pp. 1–1.

- [187] Wen, Ming, Chen, Junjie, Wu, Rongxin, Hao, Dan, and Cheung, Shing-Chi. Context-aware patch generation for better automated program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, June 2018), pp. 1–11.
- [188] Wirfs-Brock, Allen, and Terlson, Brian. ECMA-262, ECMAScript 2017 language specification, 8th edition. <https://www.ecma-international.org/ecma-262/8.0>, 2017.
- [189] Wong, Chu-Pan, Xiong, Yingfei, Zhang, Hongyu, Hao, Dan, Zhang, Lu, and Mei, Hong. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), IEEE, pp. 181–190.
- [190] Wong, W. Eric, Gao, Ruizhi, Li, Yihao, Abreu, Rui, and Wotawa, Franz. A survey on software fault localization. *IEEE Transactions on Software Engineering (TSE)* 42, 8 (2016), 707–740.
- [191] Wu, Rongxin, Zhang, Hongyu, Cheung, Shing-Chi, and Kim, Sunghun. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (2014), ACM, pp. 204–214.
- [192] Xie, Xiaoyuan, Chen, Tsong Yueh, Kuo, Fei-Ching, and Xu, Baowen. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (2013), 31.
- [193] Xin, Qi, and Reiss, Steven P. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (Santa Barbara, CA, USA, 2017), pp. 226–236.
- [194] Xiong, Yingfei, Liu, Xinyuan, Zeng, Muhan, Zhang, Lu, and Huang, Gang. Identifying patch correctness in test-based program repair. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Gothenburg, Sweden, June 2018), pp. 789–799.
- [195] Xuan, Jifeng, Martinez, Matias, Demarco, Favio, Clément, Maxime, Marcote, Sebastian Lamelas, Durieux, Thomas, Berre, Daniel Le, and Monperrus, Martin. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering (TSE)* (2016).
- [196] Yang, Xuejun, Chen, Yang, Eide, Eric, and Regehr, John. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Jose, CA, USA, 2011), pp. 283–294.
- [197] Ye, He, Martinez, Matias, Durieux, Thomas, and Monperrus, Martin. A comprehensive study of automatic program repair on the QuixBugs benchmark. In *IEEE International Workshop on Intelligent Bug Fixing (IBF)* (Hangzhou, China, Feb. 2019), pp. 1–10.
- [198] Ye, Xin, Bunescu, Razvan, and Liu, Chang. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 689–699.

- [199] Ye, Xin, Shen, Hui, Ma, Xiao, Bunescu, Razvan, and Liu, Chang. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering* (2016), ACM, pp. 404–415.
- [200] Yu, Zhongxing, Martinez, Matias, Danglot, Benjamin, Durieux, Thomas, and Monperrus, Martin. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (Feb. 2019), 33–67.
- [201] Zhai, Chengxiang. Notes on the lemur tfidf model. *Unpublished report* (2001).
- [202] Zhang, Lingming, Zhang, Lu, and Khurshid, Sarfraz. Injecting mechanical faults to localize developer faults for evolving software. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 765–784.
- [203] Zhang, Xiangyu, Gupta, Neelam, and Gupta, Rajiv. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering* (2006), ACM, pp. 272–281.
- [204] Zhong, Hao, and Su, Zhendong. An empirical study on real bug fixes. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (Florence, Italy, May 2015).
- [205] Zhou, Jian, Zhang, Hongyu, and Lo, David. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 14–24.
- [206] Zhou, Jian, Zhang, Hongyu, and Lo, David. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering* (2012), IEEE Press, pp. 14–24.
- [207] Zou, Daming, Liang, Jingjing, Xiong, Yingfei, Ernst, Michael D, and Zhang, Lu. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* (2019).