UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2020-080
The Faculty of Sciences, Technology and Medicine

Dissertation

Defence held on 07/12/2020 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg

en Informatique

by

Anil KOYUNCU
Born on 3^{rd} April 1986 in Karsiyaka, Turkey

Boosting Automated Program Repair for adoption by practitioners

Dissertation Defence Committee

Dr. Yves Le Traon, Dissertation Supervisor
Professor, University of Luxembourg, Luxembourg

Dr. Jacques Klein, Chairman
Associate Professor, University of Luxembourg, Luxembourg

Dr. Tegawendé Bissyandé, Vice Chairman
Assistant Professor, University of Luxembourg, Luxembourg

Dr. Earl T. Barr
Professor, University College London, UK

Dr. Michael Pradel
Professor, University of Stuttgart, Germany

Abstract

Automated program repair (APR) attracts huge interest from research and industry as the ultimate target in the automation of software maintenance. Towards realising this automation promise, the research community has explored various ideas and techniques, which are increasingly demonstrating that APR is no longer fictional. Although literature techniques constantly set new records in fixing a significant fraction of defects within well-established benchmarks, we are not aware of the large-scale adoption of APR in practice. Meanwhile, open-source and commercial organisations have started to reflect on the potential of integrating some automated steps in the software development cycle. Actually, the current practice has several development settings that use a number of tools to automate and systematise various tasks such as code style checking, bug detection, and systematic patching. Our work is motivated by this fact. We advocate that systematic and empirical exploration of the current practice that leverages tools to automate debugging tasks would provide valuable insights for rethinking and boosting the APR agenda towards its acceptability by developer communities. We have identified three investigation axes in this dissertation. First, **mining software repositories** towards understanding code change properties that could be valuable to guide program repair. Second, **analysing communication channels in software development** in order to assess to what extent they could be relevant in a real-world program repair scenario. Third, **exploring generic concepts of patching** in the literature for establishing a common foundation for program repair pipelines that can be integrated with industrial settings.

This dissertation makes the following contributions to the community:

- An empirical study of tool support in a real development setting providing concrete insights on the acceptance, stability, and the nature of bugs being fixed by manually-craft patches vs. tool-supported patches and manifests opportunities for improving automated repair techniques.
- A novel information retrieval based bug localisation approach that learns how to compute the similarity scores of various types of features.
- An automated mining strategy to infer fix pattern that can be integrated into automated program repair pipelines.
- A practical bug report driven program repair pipeline.

To my dear family...

Acknowledgements

Throughout my Ph.D., I have received a great deal of support and assistance from many people, and without their guidance, this dissertation would not have been possible. I would like to express my gratitude for every one of them.

First and foremost, I would like to thank my supervisor, Yves Le Traon, for giving me this great opportunity to pursue my doctoral studies in his team. I am grateful for the time he spent discussing with me new ideas and contributions.

Then, I would like to thank my day-to-day advisor, Tegawendé Bissyandé. He is the one who has introduced me to the world of automated program repair. His expertise was invaluable in formulating the research questions and methodology. His insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I am equally grateful to Jacques Klein and Dongsun Kim for their continuous support, valuable advice, and guidance.

I would like to thank Martin Monperrus for his constructive and insightful feedback on my work, as well as great advice.

I would like to thank all the members of my Ph.D. defence committee, including chairman Jacques Klein, Earl Barr, Michael Pradel, Dongsun Kim, my supervisor Yves Le Traon, and my daily adviser Tegawendé Bissyandé. It is my great honour to have them on my defence committee, and I appreciate very much their efforts to examine my dissertation and evaluate my Ph.D. work.

I would like to acknowledge my colleagues from my internship at LIP6 of Sorbonne University for their wonderful collaboration. I would particularly like to single out my supervisor at LIP6, Dr. Julia Lawall. Thank you for your patient support and for all of the opportunities, I was given to further my research.

I would like to extend my thanks to all my co-authors, including Kui Liu, Kisub Kim, Haoye Tian, Abdoul Kader Kaboreé, for their valuable discussions and collaborations.

Lastly, I would like to thank my beloved family for their endless love and continuous support.

Anil Koyuncu
University of Luxembourg
December 2020

Contents

List of figures

List of tables

1 Introduction

Contents

Fault free software is an illusion. To cope with this reality, a huge effort has been invested by the research community to increase automation in software maintenance. An ultimate automation target in software maintenance is automatic program repair (APR). APR is broadly about generating corrective patches in an automated manner in order to eliminate the bugs in programs without breaking any existing functionality. Towards achieving this ambition, the research on automated program repair has explored various ideas, algorithms, techniques for scoping, and sorting the space of patch candidates effectively and efficiently. The literature includes a broad range of techniques that use heuristics (e.g., via random mutation operations [115]), constraints solving (e.g., via symbolic execution [172]), or machine learning (e.g., via building a code transformation model [63]) to drive patch generation. The associated literature demonstrated an incredible momentum subsequent to the seminal work of Weimer et al. [237] on generate-and-validate approaches. Over the years, the community has incrementally advanced the state-of-the-art with numerous test-based approaches that have been shown effective in generating valid patches for a significant fraction of defects within well-established benchmarks [78, 123, 145, 197].

Despite this excitement in the research community, adoption by practitioners remains limited. Meanwhile, however, practitioners benefit from a number of tools to automate and systematise various tasks such as code style checking, bug detection, and systematic patching. Our work is motivated by this fact. We advocate that the exploration of practitioner's realities and expectations would facilitate the adoption of the automated program repair systems in practice. To support our endeavour, we have investigated the nature of the current practice and have observed the following aspects:

① Identifying fault locations under conditions which appropriately reflect development settings remains a largely open problem. To the best of our knowledge, most of the current state-of-the-art APR approaches [35, 39, 84, 88, 110, 111, 114, 117, 127, 132, 137, 138, 140, 155, 172, 238, 253, 255, 257] leverage test suites to perform fault localisation as test suites an affordable approximation to program specifications given the absence of formal specifications. While current test-based APR approaches would be suitable with carefully crafted benchmarks, their adoption by practitioners struggles as in practice for most development settings, many bugs are reported without the available test suite being able to reveal them.

② The intractability of the fix patterns and the generalisability (i.e., the scope of mining) of the mining strategies remain a challenge for deriving relevant patterns for program repair. Early techniques such as GenProg [117, 238] relied on simple mutation operators to drive the genetic evolution of the code. More widespread today are approaches that build on fix patterns [88] (also referred to as fix templates [135] or program transformation schemas [72]) learned from existing patches. Several APR systems [48, 72, 88, 99, 129, 131, 132, 132, 133, 135, 153, 200] in the literature discovering diverse sets of fix patterns obtained either via manual generation or automatic mining of bug-fix datasets. Manual summarisation of fix patterns is a heavy burden for APR practitioners. In addition, manual mining of fix patterns cannot enumerate the common and effective fix patterns as much as possible. Recent automatic mining techniques use frequency of code change actions [76, 241], static analysis violations [129, 196] and from Q&A posts [135] to mine fix patterns the intractability of the fix patterns. Template-based program repair systems, whether they leverage specifically pre-defined mutation operators, infer code transformations on-the-fly or rely on offline-inferred fix patterns, they generally build on data of existing code bases (preferably with a large history of code changes). If the source of mining is not appropriate (e.g., limited recurrent changes or changes associated with domain-specific bugs), the mined patterns may be irrelevant for the program that is targeted for repair. Overall, although the literature approaches can come in handy for discovering diverse sets of fix patterns, the reality is that the intractability of the fix patterns and the generalisability (i.e., the scope of mining) of the mining strategies remain a challenge for deriving relevant patterns for program repair.

③ Reliable automated repair techniques could be beneficial in a production development chain as debugging aids [221]. Consequently, it is crucial to ensure that all advancements can be measured and assessed rigorously in terms of efficiency, efficacy, and usability to perceive affordance by practitioners. Even though the automated research community has already started to reflect on the acceptability [88, 162] and correctness [210, 254] of the patches generated by APR tools, various steps, and artefacts in automated program repair techniques remain largely intractable. This intractability remains a big obstacle for transparency. In addition to transparency, approaches in the literature are often provided in monolithic tooling, which prevents extension, adaptation and even application on real-world development setting as they require substantial engineering effort for experimental adaptation.

These aspects urge to perform additional research in building automatic patch generation systems that are based on flexible, transparent and practical techniques to enable better assessment of research advancements and to facilitate the adoption of APR by software maintainers.

1.1 This thesis

In this dissertation, we propose to go back to the basics, systematically and empirically exploring the current practice of repair to provide actionable insights for, what can be repaired, how it can be repaired and when it can be repaired. We aim to offer actionable insights for rethinking and boosting the automated repair agenda towards its acceptability by developer communities.

The main objectives of this thesis are as follows:

- (a) Systematically and empirically explore the current practice of repair to provide extensive insights for, how the developer community can accept tool-supported patches (automated repair), and the automation of what kind of fixes can be readily accepted in the community.
- (b) Devise a new automated repair approach oriented towards fixing user-reported bugs under conditions which appropriately reflect development settings.
- (c) Devise a new transparent and flexible automated repair approach that builds on the concept of generic patch, that defines a unified representation/notation for specifying fix patterns (aka templates).

Concretely, towards realising these objectives, in this dissertation, we focus on:

- **Mining software repositories** towards understanding their characteristics, and explore insights on how to leverage them to facilitate program repair.
- **Analysing communication channels in software development** in order to assess to what extent they could be relevant in a real-world program repair scenario.
- **Exploring generic concepts of patching** in the literature for establishing a common foundation for program repair pipelines that can be integrated with industrial settings.

1.2 Contributions

We now summarise the contributions of this dissertation as below:

- **Impact of Tool Support in Patch Construction.** We have studied the impact of tool support in patch construction, leveraging real-world patching processes in the Linux kernel development project, to gather insights on the practice of patching as well as how the development community is currently embracing research and commercial patching tools to improve productivity in repair. Concretely, we focus on the differences between three patching processes: (1) patches crafted entirely manually to fix bugs, (2) those that are derived from warnings of bug detection tools, and (3) those that are automatically generated based on fix patterns. Our study yielded several findings about i) the acceptance of patches ii) stability of the patches iii) on the nature of bugs being fixed iv) opportunities for improving automated repair techniques in production environments.

This work has led to a research paper published in the Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017 (ISSTA 2017).

- **D&C: A Divide-and-Conquer Approach to IR-based Bug Localisation.**
  We extensively study the performance of state-of-the-art bug localisation tools, specifically focusing on investigating the query formulation (i.e., which bug report features should be compared against which features of source code files) and its importance with respect to the localisation performance. Building on insights from this study, we propose D&C, a novel IRBL approach which adaptively learns to compute the weight to associate to similarity scores of IRBL features. The training scenario builds on our findings that the various state-of-the-art localisation tools (hence the associated similarity features that they leverage) can be highly performant for specific sets of bug reports. Concretely, we leverage a gradient boosting supervised learning technique to build multi-classifiers by training on homogeneous sets of bug reports whose localisations appear to be successful with specific types of features.
  The results of this research will be soon submitted to the Springer Empirical Software Engineering Journal (EMSE).
- **iFixR: Bug Report driven Program Repair.**
  We have investigated the feasibility of automating patch generation from bug reports. To that end, we implemented iFixR; an APR pipeline variant adapted to the constraints of test cases unavailability when users report bugs. The proposed system revisits the fundamental steps, notably fault localisation, patch generation and patch validation, which are all tightly-dependent to the positive test cases in a test-based APR system. In particular, iFixR replaces classical spectrum-based fault localisation with Information Retrieval (IR)-based fault localisation. We take as input the bug report in the natural language submitted by the program user and rely on the information in this report to localise the bug positions. We make no assumptions on the availability of positive test cases that encode functionality requirements at the time the bug is discovered, and we assume only the presence of regression test cases to validate patch candidates. We further propose a strategy to prioritise patches for recommendation to developers in order to increase the probability of placing a correct patch on top of the list as in the absence of a complete test suite, we cannot guarantee that all patches that pass regression tests will fix the bug.
  This work has led to a research paper published in the Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (FSE 2019).
- **FixMiner: Mining Relevant Fix Patterns for Automated Program Repair.**
  We propose to investigate the feasibility of mining relevant fix patterns that can be easily integrated into an automated pattern-based program repair system. To that end, we propose an iterative and three-fold clustering strategy, `FixMiner`, to discover relevant fix patterns automatically from atomic changes within real-world developer fixes. The goal of `FixMiner` is to infer separate and reusable fix patterns that can be leveraged in other patch generation systems. In order to convey the full syntactic and semantic meaning of the code changes, we introduce the concept of Rich Edit Script, which is a specialised tree data structure of the edit scripts that captures the AST-level context of code changes. `FixMiner` infer patterns by discovering cluster of patches that are sharing a common representation, which is computed based on the following information encoded in Rich Edit Scripts for each round of the iteration: abstract syntax tree, edit actions tree, and code context tree. We assess the consistency of the `FixMiner` patterns with the patterns in the literature. We further demonstrate with the implementation of an automated repair pipeline that the patterns mined by `FixMiner` are relevant for generating correct patches.
  This work has led to a research papers published in Springer Empirical Software Engineering Journal (EMSE 2020).
- **FlexiRepair: Transparent Program Repair with Generic Patches.**
  Template-based program repair research needs a common ground to express fix patterns in a

standard and reusable manner. We propose to build on the concept of generic patch (also known as semantic patch), which is widely used in the Linux community to automate code evolution. We advocate that generic patches could provide at the same time a unified representation and a specification for fix patterns.

We present the design and implementation of a repair framework, FlexiRepair, that explores generic patches as the core concept. In particular, we show how concretely generic patches can be inferred and applied in a pipeline of Automated Program Repair (APR). With FlexiRepair, we address an urgent challenge in the template-based APR community to separate implementation details from actual scientific contributions by providing an open, transparent and flexible repair pipeline on top of which all advancements in terms of efficiency, efficacy and usability can be measured and assessed rigorously. Furthermore, because the underlying tools and concepts have already been accepted by a wide practitioner community, we expect FlexiRepair 's adoption by industry to be facilitated. Preliminary experiments with a prototype FlexiRepair on the IntroClass and CodeFlaws benchmarks suggest that it already constitutes a solid baseline with comparable performance to some of state of the art.

This work has led to a research paper submitted to the 43rd ACM/IEEE International Conference on Software Engineering (ICSE 2021).

1.3 Roadmap

In the remaining sections of the dissertation first, in Chapter 2 we give a brief introduction of the necessary background information and the related work in automated program repair. In Chapter 3, we present an empirical study of tool support in a real development setting, in order to assess the acceptance, stability and the nature of the bugs being fixed with manually crafted patches against patches generated with tool-support. In Chapter 4, we focus on mining software repositories towards guiding program repair, and we present an automated mining approach to infer fix patterns that can be integrated into automated program repair pipelines. In Chapter 5, we focus on communication channels in software development, especially bug reports, and present a novel information retrieval based bug localisation approach and a practical bug report driven program pipeline. In Chapter 6, we propose an open, transparent and flexible program repair pipeline that builds on the concept of generic patches allowing to express fix patterns in a standard, reusable manner. Finally, in Chapter 7, we conclude this dissertation and discuss some potential future works.

2 Background & Related Work

In this chapter, we provide the preliminary details that are necessary to understand the purpose, techniques and key concerns of the various research studies that we have conducted in this dissertation. Mainly, we revisit the literature of automated program repair, focusing on fault localisation, patch generation and patch validation, respectively.

Contents

2.1 Automated Program Repair

The research on automated program repair has explored various ideas, algorithms, techniques for traversing a search space of patch candidates that are generated by applying change operators to the buggy program code. Depending on how a technique conducts the search and constructs the patches, the literature includes a broad range of techniques that use *heuristics-based* [76, 88, 133, 238], *constraint-based* [156, 172, 255] and *learning-aided* [63, 137, 140] following the taxonomy proposed by Le Goues *et al.* [118]. Most of these techniques take a buggy program and a correctness criterion (often test suites as they represent an affordable approximation to program specifications) as their input. Generally, they follow a set of activities starting with i) a fault localisation step that identifies code locations that are likely to be buggy ii) a patch generation step that conducts the search and construction of the patch candidates and iii) a patch validation step that checks whether the proposed fix actually corrects the bug.

2.1.1 Fault Localisation

Fault localisation (FL) research focuses on developing automated techniques to identify program entities (such as source code files, methods, and statements) that are likely to contain the defect. FL techniques leverage dynamic analysis, runtime information and static analysis to identify a potential buggy program element. Depending on what FL techniques leverage as the source of information to identify the suspicious program entity, they can be classified as follows:

Spectrum-based fault localisation (SBFL) techniques use test coverage information [3, 65, 251], mutation-based fault localisation (MBFL) techniques use test results collected from mutating the program [164,182] (dynamic) program slicing techniques use the dynamic program dependencies [7,194], stack trace analysis techniques use error messages [247, 250], predicate switching techniques use test results from mutating the results of conditional expressions [273], information-retrieval based fault localisation (IRFL) techniques use bug report information [199, 242, 247, 267, 276], and history-based techniques use the development history to identify the suspicious program elements that are likely to be defective [94,191]. Recently, the literature leverages machine learning and deep learning techniques to perform fault localisation. Ye et al. [263] proposed a learning-to-rank approach to bug localisation based features representing the degree of suspiciousness. Kim et al. [89] dealt with bug report quality to improve bug localisation with a two-phase model focusing on high-quality bug reports. Lam et al. presented HyLoc [103] and DNNLoc [104] that use deep neural networks to learn relevancy between tokens in bug reports and code elements in the source code. In this dissertation, we propose a novel IRFL approach, D&C, which adaptively learns to compute the weight to associate to similarity scores of features from sets of bug reports whose localisations appear to be successful with specific types of features.

In the scope of automated program repair, most APR systems use SBFL [6, 36, 62, 63, 76, 99, 127, 132, 133, 140, 152, 154, 200, 241, 253, 255]. SBFL leverages execution coverage information of passing and failing test cases and a formula such as Ochiai [3] to spot the bug positions. In the literature, APR systems often rely on a testing framework [35, 72, 114, 200] such as GZoltar [31], and a spectrum-based fault localisation formula [249, 258, 274], such as Ochiai [3]. The popularity of Ochiai is backed up by empirical evidence on its effectiveness to help localise faults in object-oriented programs as highlighted by several fault localisation studies [182, 216, 251, 258]. Pearson et al. [186] has even shown that Ochiai outperforms current state-of-the-art ranking metrics, or at least offers similar performance measures.

Recently, some APR systems start to leverage supplementary information to assist the fault localisation step and improve accuracy. For example, HDRepair [114], JAID [35] and SketchFix [72] assume that the faulty methods are known: the fault localisation step, therefore, focuses on ranking the lines inside the method, thus leaving out noisy statements that other APR tools are considering.

ssFix [253] prioritises statements from the stack trace of crashed programs that are executed before those statements that are ranked by the FL tool. ACS [255] uses predicate switching [273] and refines the suspicious code locations list. SimFix [76] applies a test case purification approach to improve the accuracy of fault localisation step before patch generation. Liu et al. [131] has investigated the fault localisation impact on the repair performance of APR systems highlighting the potential biases due to the elision of assumptions and tweaks details while presenting the results of repair tools.

Meanwhile, a few studies [20, 126] discussed the possibility to leverage bug reports in the context of automated program repair. To the best of our knowledge, Liu et al. [126] proposed the most advanced study in this direction. However, their R2Fix approach does not use any fault localisation, it rather focuses on bug reports [126, page 283] which explicitly include localisation information. In this dissertation, we have investigated the feasibility of leveraging bug reports in the context of the APR. Concretely, we propose an APR system, `iFixR` that is driven by bug reports, that replaces classical spectrum-based fault localisation with Information Retrieval (IR)-based fault localisation.

2.1.2 Patch Generation

The literature includes a broad range of techniques that use heuristics (e.g., via random mutation operations [115]), constraints solving (e.g., via symbolic execution [172]), or machine learning (e.g., via building a code transformation model [63]) to drive patch generation.

Heuristic-based repair approaches employ a generate-and-validate methodology, which constructs and iterates over a search space of syntactic program modifications [118] that are then validated by running the modified program on the provided set of test cases. The search space denotes a set of considered modifications (also referred to as patch candidates [88]) for a buggy program that is generated by a dedicated repair approach. The validation proceeds with the evaluation of patch candidates by executing the provided set of test cases. Once a patch candidate can make a buggy program pass all tests, it is considered as the patch for the buggy program. Notable APR tools for Java programs include jGenProg [152], GenProg-A [271], ARJA [271], RSRepair-A [271], SimFix [76], jKali [152], Kali-A [271], jMutRepair [152], HDRepair [114], PAR [88], S3 [110], ELIXIR [200], SOFix [135], CapGen [241].

Constraint-based repair approaches proceed with a methodology different from heuristic-based repair, which constructs repair constraints that will be used to select donor code for the patch generation [63]. The constraints describe the code fragments that should satisfy the variable types, values or behaviour specified by the constraints. Such code fragments are returned as the potential matches, which further be synthesised into patch candidates with the specific functions. For example, symbolic execution approaches extract properties about the function to be synthesised; these properties constitute the repair constraints. Solutions to the repair constraints can be obtained by constraint solving or other search techniques. Nopol [257], DynaMoth [49] ACS [255], Cardumen [153], SemFix [172] and Angelix [156] are notable constraint-based repair approaches.

Learning-based repair approaches explore the advanced machine learning technique, especially deep learning technique, to boosting program repair [118]. To date, learning-based repair has been exploited in three ways: 1) learning models of correct code that are used to prioritise the patch candidates in terms of the correctness [140]; 2) learning code transformation patterns that summarise how human-written patches change buggy code into correct code [27, 137], where patterns can be further used to generate patch candidates; and 3) learning to improve the repair process and training models for end-to-end repair, where models are leveraged to predict the correct code for the given buggy code without using any other explicitly provided context information [36, 228] and learning the context of the code surrounding a fix [121].

More widespread today are approaches that build on fix patterns [88] (also referred to as fix templates [135] or program transformation schemas [72]) learned from existing patches which could be grouped into heuristic-based repair as the main process of fix pattern-based repair is consistent with heuristic-based repair. Several APR systems [48, 72, 88, 99, 129, 131, 132, 132, 133, 135, 153, 200] implement this strategy by using diverse sets of fix patterns obtained either via manual generation or automatic mining of bug-fix datasets. Depending on how a technique obtains fix patterns, it can be categorised into four groups: manual summarizing, pre-definition, frequency and mining.

1. *Manual Summarizing: Pan et al. [181] manually identified 27 fix patterns from patches of five Java projects to characterize the fix ingredients of patches. Kim et al. [88] manually summarised 10 fix patterns from 62,656 human-written patches collected from Eclipse JDT.*
2. Pre-definition: Durieux et al. [48] pre-defined 9 repair actions for null pointer exceptions by unifying the related fix patterns proposed in previous studies [44, 85, 141]. On the top of *PAR* [88], Saha et al. [200] further defined 3 new fix patterns to improve the repair performance. Hua et al. [72] proposed an APR tool with six pre-defined so-called code transformation schemas. Xin and Reiss [253] proposed an approach to fixing bugs with 34 pre-defined code change rules at the AST level.
3. *Frequency: Besides formatted fix patterns, researchers [76, 241] also explored to automate program repair with code change instructions (at the abstract syntax tree level) that are frequently recurring in existing patches [76, 130, 151, 240, 275]. The strategy is then to select the top-*n *most frequent code change instructions as fix ingredients to synthesize patches.*
4. Mining: Long et al. [137] proposed Genesis, to infer fix patterns for three kinds of defects from existing patches. Liu and Zhong [135] explored fix patterns from Q&A posts in Stack Overflow. Liu et al. [129] and Rolim et al. [196] proposed to mine fix patterns from static analysis violations from FindBugs and PMD respectively.

This dissertation also focuses on fix pattern-based program repair. We consider the predefined set of patterns used by literature APR systems, as well as we investigate the feasibility of an automated approach to mine relevant and actionable fix patterns that can be easily integrated into an automated pattern-based program repair system.

In the last decade, most proposed techniques in the literature present repair pipelines where patch candidates are generated then validated against a program specification, generally a (weak) test suite. We refer to them as *generate-and-validate* test-suite based repair approaches. The genetic programming-based approach proposed by Weimer *et al.* [238], as well as follow-up works, appeared only valid for hypothetical use cases as the assumption that *test cases are readily available* still does not hold in practice [15, 95, 187]. Nevertheless, in the last couple of years, two independent reports have illustrated the use of literature techniques in actual development flows: in the open source community, the Repairnator project [230] has successfully demonstrated that automated repair engines can be reliable: open source maintainers accepted and merged patches which were suggested by an APR bot. At the premises of Facebook, the SapFix repair system has been reported to be part of the continuous integration pipeline [148] while Getafix was used there at large scale [12]. In this dissertation, our aim is to devise automated repair approaches facilitating the adoption of the automated program repair systems in practice. In this direction, we focus not only improvements on the current *generate-and-validate* repair approaches, but also we devise a new automated repair approach oriented towards fixing user-reported bugs under conditions which appropriately reflect development settings.

2.1.3 Patch Validation

The patch validation activity checks whether the proposed patch candidates actually fixes the bug. So far, most of the generate-and-validate techniques establish the adequateness of the solutions by running the available test cases, that is, if a program in the set of the candidate solutions passes all the available test cases, the program is returned to the developer as a possible fix. The assessment of APR approaches in the literature generally attempts to provide information on the number of bugs for which APR tool can generate a patch that makes the buggy program pass all the test cases [88, 114, 152, 238, 257]. However, analysing patch correctness was largely ignored or unconcerned in the community until the analysis study of patch correctness conducted by Qi *et al.* [190]. Their systematic analysis of the patches reported by three generate-and-validate program repair systems (i.e., GenProg, RSRepair and AE) shown that the overwhelming majority of the generated patches are not correct but just overfit the test inputs in the test suites of buggy programs. In another study, Smith *et al.* [210] uncover that patches generated with lower coverage test suites overfit more. Actually, these overfitting patches often simply break under-tested functionalities, and some of them even make the "patched" program worse than the unpatched program. Since then, the overfitting issue has been widely studied in the literature.

Eventually, to fairly assess the performance on fixing real bugs of APR tools, the number of bugs for which a correct (i.e., it is semantically equivalent to the patch that the program developer accepts for fixing the bug) patch is generated appeared to be a more reasonable metric than the mere number of plausible patches [255]. This metric has since then become standard among researchers, and is now widely accepted in the literature for evaluating APR tools [35, 72, 76, 127, 132, 133, 135, 200, 241]. Based on data presented with such metric, researchers explicitly rank the APR systems, and use this ranking as a validation of new achievements in program repair. However, this has been a manual effort based on a recurrent criterion: a plausible patch is considered as correct when it is semantically similar to the developer's patch in the benchmark.

Therefore, researchers have started to focus some effort in automating the identification of patch correctness [254]. Le *et al.* [112] revisit the overfitting problem in semantics-based APR systems. Le *et al.* [113] further assess the reliability of authors and automated annotations in assessing patch correctness. They recommend making publicly available to the community the patch correctness evaluations of the authors. Yang and Yang [259] explore the difference between the runtime behaviour of programs patched with developer's patches and those by APR-generated plausible patches. They unveil that the majority of the APR-generated plausible patches leads to different runtime behaviour compared to correct patches. Liu *et al.* [134] propose to unveil the implicit rules that researchers use to make the decisions on correctness.

In recent literature, researchers are investigating to predict the correctness of patches. One of the first explored research directions relied on the idea of augmenting test inputs, i.e., more tests need to be proposed. Yang *et al.* [260] design a framework to detect overfitting patches. This framework leverages fuzz strategies on existing test cases in order to automatically generate new test inputs. In addition, it leverages additional oracles (i.e., memory-safety oracles) to improve the validation of APR-generated patches. In a contemporary study, Xin and Reiss [252] also explored to generate new test inputs, with the syntactic differences between the buggy code and its patched code, for validating the correctness of APR-generated patches. As complemented by Xiong *et al.* [254], they proposed to assess the patch correctness of APR systems by leveraging the automated generation of new test cases and measuring behaviour similarity of the failing tests on buggy and patched programs.

Through an empirical investigation, Yu *et al.* [270] summarised two common overfitting issues: incomplete fixing and regression introduction. To assist alleviating the overfitting issue for synthesis-based APR systems, they further proposed `UnsatGuided` that relies on additional generated test cases to strengthen patch synthesis, and thus reduce the generation of incorrect overfitting patches. Ye *et al.* [261] propose ODS, an overfitting detection system, that learns an ensemble probabilistic model for classifying and ranking potentially overfitting patches.

In a recent work, Csuvik *et al.* [40] exploit the textual and structural similarity between the buggy code and the APR-patched code with two representation learning models (BERT [43] and Doc2Vec [107]) by considering three patch code representation (i.e., source code, abstract syntax tree and identifiers). Their results show that the source code representation is likely to be more effective in correct patch identification than the other two representations, and the similarity-based patch validation can filter out incorrect patches for APR tools. Tian *et al.* [224] focus on assessing representation learning techniques for predicting correctness of patches generated by program repair tools.

In this dissertation, we mainly follow the metric of numbers of plausible/correct patches to perform the evaluation of our proposed APR systems. Additionally, we investigate the feasibility of performing patch validation with regression testing (relying only on past test cases, which were available in the code base, when the bug is reported) and propose patch recommendation for validation by developers.

3 An Empirical Study of Patching in Practice

In this work, we investigate the practice of patch construction in the Linux kernel development, focusing on the differences between three patching processes: (1) patches crafted entirely manually to fix bugs, (2) those that are derived from warnings of bug detection tools, and (3) those that are automatically generated based on fix patterns. With this study, we provide to the research community concrete insights on the practice of patching as well as how the development community is currently embracing research and commercial patching tools to improve productivity in repair. In particular, we investigate the extent of the acceptance of bug finding and patch application tools in a production environment, and study the opportunities of automation that the automated repair community can explore.

This chapter is based on the work published in the following research paper:

- A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017

Contents

3.1 Overview

Patch construction is a key task in software development. In particular, it is central to the repair process when developers must engineer change operations for fixing the buggy code. In recent years, a number of tools have been integrated into software development ecosystems, contributing to reducing the burden of patch construction. The process of a patch construction indeed includes various steps that can more or less be automated: bug detection tools, for example, can help human developers characterise and often localise the piece of code to fix, while patch application tools can systematise the formation of concrete patches that can be applied within an identified context of the code.

Tool support, however, can impact patch construction in a way that may influence acceptance or that focuses the patches to specific bug kinds. The growing field of automated repair [88, 117, 156, 172], for example, is currently challenged by the nature of the patches that are produced and their eventual acceptance by development teams. Indeed, constructed patches must be applied to a code base and later maintained by human developers.

This situation raises the question of the acceptance of patches within a development team, with regards to the process that was relied upon to construct them. The goal of our study is therefore to identify different types of patches written by different construction processes by exploring patches in a real-world project, to reflect on how the program repair is conducted in current development settings. In particular, we investigate how advances in static bug detection and patch application have already been exploited to reduce human efforts in repair.

We formulate research questions for comparing different types of patches, produced with varying degrees of automation, to offer to the community some insights on i) whether tool-supported patches can be readily adopted, ii) whether tool-supported patches target specific kinds of bugs, and iii) where further opportunities lie for improving automated repair techniques in production environments.

In this work, we consider the Linux operating system development since it has established an important code base in the history of software engineering. Linux is furthermore a reliable artefact [74] for research as patches are validated by a strongly hierarchical community before they can reach the mainline code base. Developers involved in Linux development, especially maintainers who are in charge of acknowledging patches, have relatively extensive experience in programming. Linux's development history constitutes valuable information for repair studies as a number of tools have been introduced in this community to automate and systematise various tasks such as code style checking, bug detection, and systematic patching. Our analysis unfolds as an empirical comparative study of three patch construction processes:

- **Process H:** In the first process, developers must rely on a bug report written by a user to understand the problem, locate the faulty part of source code, and manually craft a fix. We refer to it as *Process H*, since all steps in the process appear to involve **H**uman intervention.
- **Process DLH:** In the second process, static analysis tools first scan the source code and report on lines which are likely faulty. Fixing the reported lines of code can be straightforward since the tools may be very descriptive on the nature of the problem. Nevertheless, dealing with static debugging tools can be tedious for developers with little experience as these tools often yield too many false positives. We refer to this process as *Process DLH*, since **D**etection and **L**ocalisation are automated but **H**uman intervention is required to form the patch.
- **Process HMG:** Finally, in the third process, developers may rely on a systematic patching tool to search for and fix a specific bug pattern. We refer to this process as *Process HMG*, since **H**uman input is needed to express the bug/fix patterns which are **M**atched by a tool to a code base to **G**enerate a concrete patch.

We ensure that the collected dataset does not include patch instances that can be attributed to more than one of the processes described above. Our analyses have eventually yielded a few implications for future research:

Acceptance of patches: development communities, such as the Linux kernel team, are becoming aware of the potential of tool support in patch construction i) to gain time by prioritising engineering tasks and ii) to attract contributions from novice developers seeking to join a project.

Kinds of bugs: Tool-supported patches do not target the same kinds of bugs as manual patches. However, we note that patches fixing warnings outputted by bug detection tools are already complex, requiring several change operations over several lines, hunks and even files of code.

Opportunities for automated repair: We have performed preliminary analyses which show that bug detection tools can be leveraged as a stepping stone for automated repair in conjunction with patch generation tools, to produce patches that are consistent with human patches (for maintenance), correct (derived from past experience of fixing a specific bug type) and thus likely to be rapidly accepted by development teams.

3.2 Background

Linux is an open-source operating system that is widely used in environments ranging from embedded systems to servers. The heart of the Linux operating system is the Linux kernel, which comprises all the code that runs with kernel privileges, including device drivers and file systems. It was first introduced in 1994, and has grown to 14.3 million lines of C code with the release of Linux 4.8 in Oct. 2016.[1] All data used in this work are related to changes propagated to the mainline code base until Oct. 2, 2016[2].

A recent study has shown that, for a collection of typical types of faults in C code, the number of faults is staying stable, even though the size of the kernel is increasing, implying that the overall quality of the code is improving [179]. Nevertheless, ensuring the correctness and maintainability of the code remains an important issue for Linux developers, as reflected by discussions on the kernel mailing list [214].

Development Model of Linux

The Linux kernel is developed according to a hierarchical open source model referred to as Benevolent dictator for life (BDFL) [245], in which anyone can contribute, but ultimately all contributions are integrated by a single person, Linus Torvalds. A Linux kernel maintainer receives patches related to a particular file or subsystem from developers or more specialised maintainers. After evaluating and locally committing them, he/she propagates them upwards in the maintainer hierarchy eventually towards Linus Torvalds.

Finally, Linux developers are urged to “solve a single problem per patch”[3], and maintainers are known to enforce this rule as revealed by discussions on contributors’ patches in the Linux Kernel Mailing List (LKML) [214] archive.

Patching and Repair in Linux

Recently, the development and maintenance of the Linux kernel have become a massive effort, involving a huge number of people. 1,731 distinct commit authors have contributed to the development of

[1] Computed with David A. Wheeler’s ‘SLOCCount’.

[2] Kernel’s Git HEAD commit id is `c8d2bc9bc39ebea8437fd974fdbc21847bb897a3`.

[3] see `Documentation/SubmittingPatches` in linux tree.

Linux 4.8[4]. The patches written by these commit authors are then validated by the 1,142 *maintainers* of Linux 4.8[5], who are responsible for the various subsystems.

Since the release of Linux 2.6.12 in June 2005, the Linux kernel has used the source code management system `git` [50]. The current Linux kernel git tree [227] only goes back to Linux 2.6.12, and thus we use this version as the starting point of our study. Between Linux 2.6.12 and Linux 4.8 there were 616,291 commits, by 20,591 different developers[6]. These commits are retrievable from the git repository as *patches*. Basically, a patch is an extract of code, in which lines beginning with `-` are to be removed lines beginning with `+` are to be added.

The Linux kernel community actively uses the Bugzilla [86] issue tracking system to report and manage bugs. As of November 2016, over 28 thousands bug reports were filed in the kernel tracking system, with about 6,000 marked as highly severe or even blocking.

The Linux community has also built, or integrated, a number of tools for improving the quality of its source code in a systematic way. For example, The mainline code base includes the coding style checker *checkpatch*, which was released in July 2007, in Linux 2.6.22. The use of checkpatch is supported by the Linux kernel guidelines for submitting patches[7], and checkpatch has been regularly maintained and extended since its inception. Sparse [244] is another example of the tools built by Linus Torvalds and colleagues to enforce type checking.

Commercial tools, such as Coverity [217], also often help to fix Linux code. More recently, researchers at Inria/LiP6 have developed the Coccinelle project [125] for Linux code matching and transformation. Initially, the project was designed to help developers perform collateral evolutions [177]. It is now intensively used by Linux developers to apply fix patterns to the whole code base.

3.3 Methodology

Our objective is to empirically check the impact of tool support in the patch construction process in Linux. To achieve this goal, we must collect a large, consistent and clean set of patches constructed in different processes. Specifically, we require:

(1) patches that have been a-priori manually prepared by developers based on the knowledge of a potential bug, somewhere in the code. For this type of patches, we assume that a user may have reported an issue while running the code. In the Linux ecosystem, such reporters are often kernel developers.

(2) patches that have been constructed by using the output of bug finding tools, which are integrated into the development chain. We consider this type of patches to be tool-supported, as debugging tools often provide reliable information on what the bug is (hence, how to fix it) and where it is located.

(3) patches that have been constructed, by a tool, based fully on change rules. Such fixes, validated by maintainers, are actually based on templates of fix patterns which are used to i) match (i.e., locate) incorrect code in the project and ii) generate a corresponding concrete fix.

[4] Obtained using `git log v4.7..v4.8 | grep ^Author | sort -u | wc -l`, without controlling for variations in names or email addresses.

[5] Obtained using `grep ^M: MAINTAINERS | sort -u | wc -l` without controlling for variations in names or email addresses.

[6] Again, we have not controlled for variations in names or email addresses.

[7] `Documentation/SubmittingPatches` in the Linux tree.

3.3.1 Dataset Collection

To collect patches constructed via Process H, hereafter referred to as ***H patches***, we consider patches whose commits are explicitly linked to a bug report from the kernel bugzilla tracking system and any other Linux distributions bug tracking systems. We consider that such patches have been engineered manually after careful consideration of the report filed by a user, and often after a replication step where developers dynamically test the software.

Until Linux 4.8, we have found 5,758 patches fixing defects described in bug reports. Unfortunately, for some of the patches, the link to its bug report provided in the commit log was not accessible (e.g., because of restriction in access rights of some Redhat bug reports or because the web page was no longer live). Consequently, we were able to collect 4,417 bug patches corresponding to a bug report (i.e., $\sim 77\%$ of H patches). Table 3.1 provides statistics on the bugs associated with those patches.

Table 3.1: Statistics on H patches in Linux Kernel.

Severity	# reports	# patches
Severe	965	1,052
Medium	2,961	3,163
Minor	138	136
Enhancement	47	66
Total	4,111	4,417

First, we note that the severity of most bugs (2,961, i.e., 72.0%) is medium, and H patches have fixed substantially more severe bugs (965, i.e., 23.5%) than minor bugs (138, i.e., 3.3%). Only 47 (1.1%) bug reports represent mere enhancements. Second, exploring the data shows that there is not always a 1 to 1 relationship between bug reports and patches: a bug report may be addressed by several patches, while a single patch may relate to several bug reports. Nevertheless, we note that 4,270 out of 5,265 (i.e., 89%) patches address a single bug report. Third, a large number of unique developers (1,088 out of 18,733= 6.95%) have provided H patches to fix user bug reports. Finally, H patches have touched about 17% (= 9,650/57,195) of files in the code base. Overall, these statistics suggest that the dataset of H patches is diverse as they are indeed written by a variety of developers to fix a variably severe set of bugs spread across different files of the program.

We identify patches constructed via Process DLH, hereafter referred to as ***DLH patches***, by matching in commit logs messages on the form "`found by <tool>`"[8] where `<tool>` refers to a tool used by kernel developers to find bugs. In this work, we consider the following notable tools, for static analysis:

- *checkpatch*: a coding style checker for ensuring some basic level of patch quality.
- *sparse*: an in-house tool for static code analysis that helps kernel developers to detect coding errors based on developer annotations.
- *Linux driver verification (LDV) project* : a set of programs, such as the Berkeley Lazy Abstraction Software verification Tool (BLAST) that solves the reachability problem, dedicated to improving the quality of kernel driver modules.
- *Smatch*: a static analysis tool.
- *Coverity*: a commercial static analysis tool.
- *Cppcheck*: an extensible static analysis tool that feeds on checking rules to detect bugs.

and for dynamic analysis:

[8] We also use "`generated by <tool>`" since the commit authors also often refer to warnings as "generated by" a given tool.

- *Strace*: a tracer for system calls and signals, to monitor interactions between processes and the Linux kernel.
- *Syzkaller*: a supervised, coverage-guided Linux syscall fuzzer for testing untrusted user input.
- *Kasan*: the Linux Kernel Address SANitizer is a dynamic memory error detector for finding use-after-free and out-of-bounds bugs.

After collecting patches referring to those tools, we further check that commit logs include terms "bug" or "fix", to focus on bug fix patches. Table 3.2 provides details on the distribution of patches produced based on the output of those tools.

Table 3.2: Statistics on DLH patches in Linux Kernel.

Tool	# patches	Tool	# patches
checkpatch	292	sparse	68
LDV	220	smatch	39
coverity	84	cppcheck	14
strace	4	syzkaller	7
kasan	1		

Checkpatch and the *Linux driver verification project* tools are the most mentioned in commit logs. The *Coverity* commercial tool and the *sparse* internal tool also helped to find and fix dozens of bugs in the kernel. Finally, we note that static tools are more frequently referred to than dynamic tools.

HMG patches in Linux are mainly carried out by Coccinelle, which was originally designed to document and automate collateral evolutions in the kernel source code [177]. Coccinelle is built on an approach where the user guides the inference process using patterns of code that reflect the user's understanding of the conventions and design of the target software system [106].

Static analysis by Coccinelle is specified by developers who use control-flow sensitive concrete syntax matching rules [28]. Coccinelle provides a language, SmPL[9], for specifying search and transformations referred to as *semantic patches*. It also includes a transformation engine for performing the specified semantic patches. To avoid confusion with semantic patches in the context of automated repair literature, we will refer to Coccinelle-generated patches as *SmPL patches*.

```
@@
expression E;
constant c;
type T;
@@
-kzalloc(c * sizeof(T), E)
+kcalloc(c, sizeof(T), E)
```

(a) Example of SmPL templates.

```
void main(int i)
{

    kzalloc(2 * sizeof(int), GFP_KERNEL);
    kzalloc(sizeof(int) * 2, GFP_KERNEL);

}
```

(b) C code matching the template on the left. (iso-kzalloc.c).

Figure 3.1: Illustration of SmPL matching and patching.

Figure 4.15 illustrates an SmPL patch example. This SmPL patch is aimed at changing all function calls of *kzalloc* to *kcalloc* with a reorganisation of call arguments. For more details on how SmPL patches are specified, we refer the reader to the project documentation[10]. Figure 3.2 represents the concrete Unix diff generated by Coccinelle engine and which is included in the patch to forward to mainline maintainers.

In some cases, the fix is not directly implemented in the SmPL patch (which is then referred to as SmPL *match*). Nevertheless, since each bug pattern must be clearly defined with SmPL, the associated fix is straightforward to engineer. Overall, we have collected 4,050 HMG patches mentioning "coccinelle" or "semantic patch" and applied to C code[11].

[9] Semantic Patch Language.

[10] `http://coccinelle.lip6.fr/documentation.php`

[11] We have controlled with a random subset of 100 commits that this grep-based approaches yielded indeed only relevant patches constructed by Coccinelle.

```
diff =
--- iso-kzalloc.c
+++ /tmp/cocci-output-52882-062587-iso-kzalloc.c
@@ -1,7 +1,7 @@
 void main(int i)
 {
-   kzalloc(2 * sizeof(int), GFP_KERNEL);
-   kzalloc(sizeof(int) * 2, GFP_KERNEL);
+   kcalloc(2, sizeof(int), GFP_KERNEL);
+   kcalloc(2, sizeof(int), GFP_KERNEL);
 }
```

Figure 3.2: Patch derived from the SmPL template in Figure 4.15a.

3.3.2 Research Questions

We now enumerate and motivate our research questions in the context of the three processes of patch construction:

RQ1 *How does the developer community react to the introduction of bug detection and patch application tools?*
With this research question, we check that the temporal distributions of patches in each patch construction process are in line with the upstream discussions for accepting patches. Such discussions may shed light on the proportions of tool-supported patches that are pushed by developers but that never get into the code base.

RQ2 *Who is using bug detection and patch application tools?*
In this research question, we investigate the profile of patch authors in the different patch construction processes.

RQ3 *What is the impact of patch construction process in the stability of patches?*
We investigate the stability, i.e., whether or not the patch is reverted after being propagated in the mainline tree, of accepted patches to highlight the reliability of each patch application tool within the community.

RQ4 *Do the patch construction processes target the same kind of bugs?*
We approximate the categorisation of bugs with two metrics related to (1) the locality of the fixes as well as (2) the nature and number of change operators of the patch.

3.4 Empirical Study Findings

3.4.1 Descriptive Statistics on the Data

We first provide statistics on how the different patch construction processes are used by developers over time and across project modules. Temporal distribution of patches may shed some light on the adoption of a patch construction process by kernel maintainers. Spatial distributions on the other hand may highlight the acceptance of a process based on the type (i.e., to some extent the critical nature) of the code to fix.

Temporal distribution of patches. We compute the temporal distribution of patches since Linux 2.6.12 (June 2005) until Linux 4.8 (October 2016) and outline them in Figure 3.3. Note that although Linux 2.6.12 was released in June 2005, a few commit patches in the code base pre-date this release date.

Overall, H patches are consistently applied over time with approximately 50 fixes per month. DLH patches have been very slow to take up. Indeed, the number of patches built based on bug finding tools has been narrow for several years, with a slight increase in recent years, partly due to the improvements made for reducing false positives. Finally, HMG patches have rapidly increased and now account for a significant portion of patches propagated to the mainline code base.

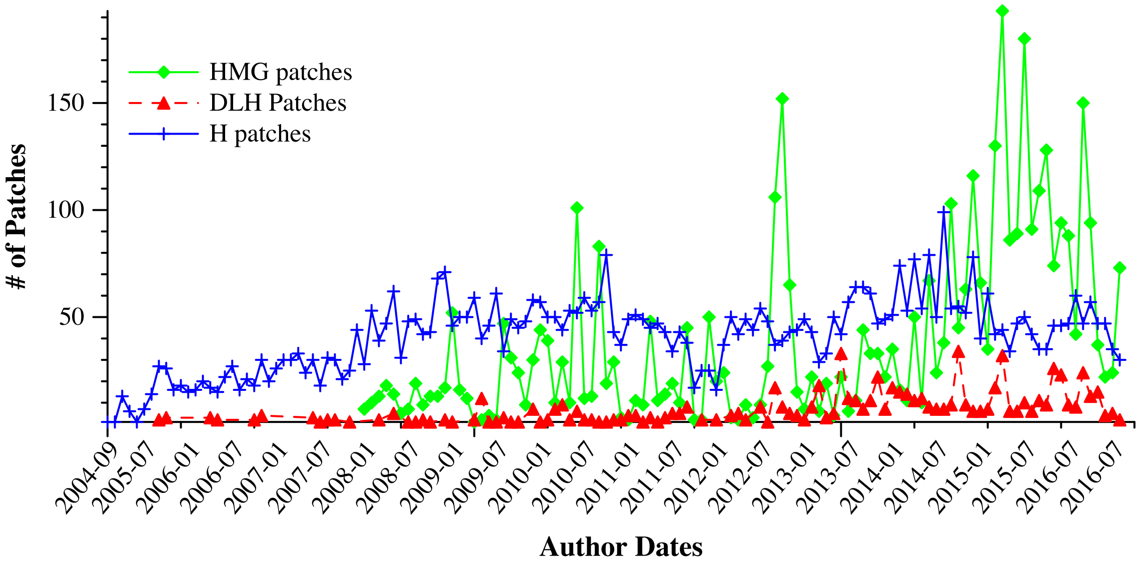


Figure 3.3: Temporal distributions of patches.

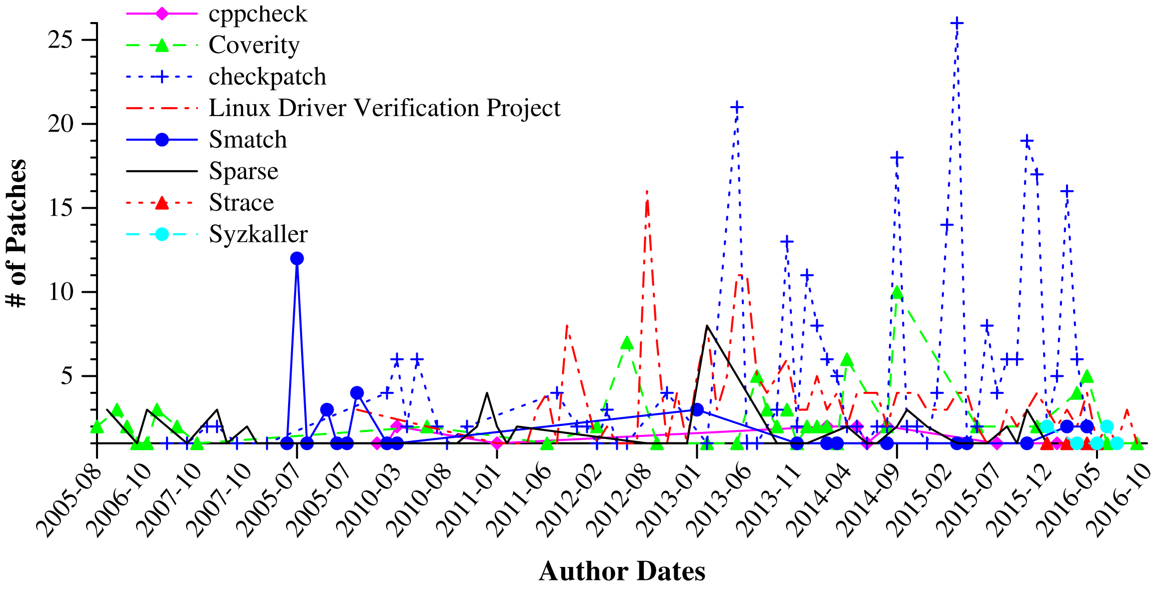
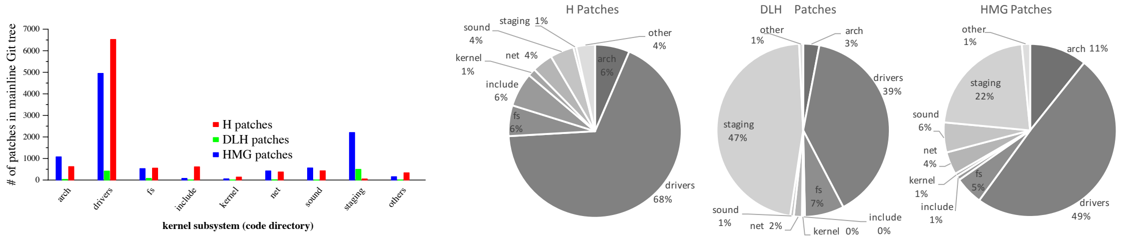


Figure 3.4: Temporal distributions of DLH patches broken down by tool.

Figure 3.4 represents the detailed temporal evolutions of DLH patches. Checkpatch, after a slow adoption, is now commonly used, followed by Coverity, which regularly contributes to fix vulnerabilities and common operating system errors. Linux driver verification project tools and Smatch find fewer issues in mainline code base; such tools are indeed extensively used by developers before code is committed in the code base.

Spatial distribution of patches. We compute the spatial distribution of patches across Linux sub-systems. Linux Kernel's code is split into several folders, each roughly containing all code related to a specific sub-system such as file systems, device drivers, architectures, networking, etc. We investigate the scenarios of patches with regards to the folders where the files are changed and the results are shown in Figure 3.5. Most patches are targeted to device drivers code, and code in early development (i.e., in *staging/*[12]) that is not yet part of the running kernel. It is noteworthy that

[12] *staging* is a sub-directory of *drivers* and contains code that does not yet meet kernel coding standards. We thus separate its statistics from statistics of *drivers*.



(a) Number of patches committed by each patch process to Linux's mainline code base.

(b) Percentage of patches per subsystem.

Figure 3.5: Spatial distribution of patches.

header code (*include/*), core kernel code (*kernel/*), and to some extent file system code (*fs/*), which have been extensively tested over the years, remain repaired mainly in an all-human process.

Driver code in general, and *drivers/staging/* code, in particular, appear to be the place where tool support is most prevalent. Percentages distribution in Figure 3.5b shows that half (46%) of DLH patches are targeted at *staging* code. 39% of DLH patches are applied to driver code. Several studies [38, 179, 180] have already shown that driver and staging code contained most kernel errors identified by static analysis tools. Similarly, HMG patches are applied in a large majority in drivers code and staging code.

3.4.2 Acceptance of Patches (RQ1)

We investigate the reaction of the developer community to the introduction of bug finding and patch application tools. To that end, we explore, first, the delays in integrating commits, then, the gaps between the number of patches proposed to the Linux community and those that are finally integrated.

Delay in commit acceptance. Kernel patches are change suggestions proposed by developers to maintainers who often need time to review them before propagating the changes to the mainline code base. Thus, depending on several factors — including the criticality of the bug, complexity of the fix, reliability of the suggested fix, and patch quality — there can be a more or less significant delay in commits.

We compute a delay in commit acceptance as the time difference between the author contribution date and the commit date (i.e., when the maintainer propagated the patch to mainline tree). Figure 3.6 shows the distribution of delays in the three different patch construction processes. Overall, H patches appear to be more[13] rapidly propagated (median = 2 days) than DLH (median = 4 days) and HMG patches (median = 4 days).

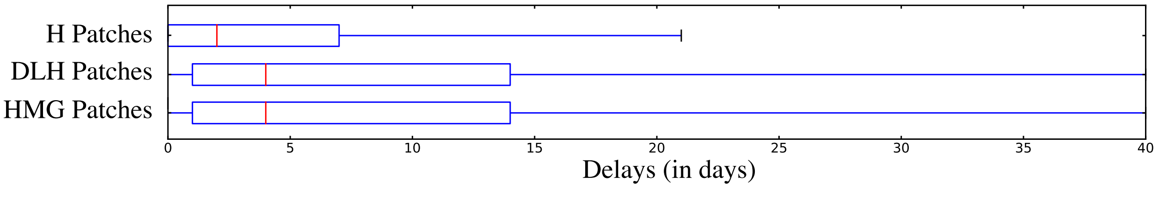
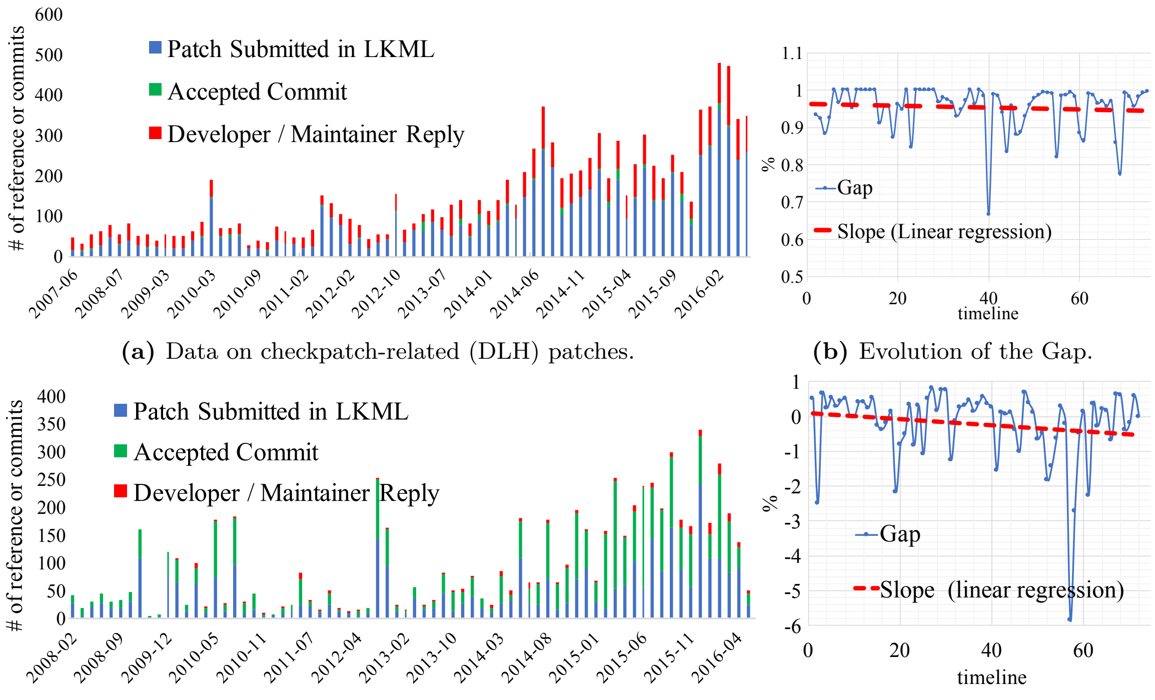


Figure 3.6: Delay in commit acceptance.

Gaps between discussion and acceptance trends. A patch represents the conclusion of an email exchange between the patch author and the relevant maintainers about the correctness of

[13] We have checked with the Mann-Whitney Wilcoxon test that the difference between delay values is statistically significant.

the proposed change. As the discussion takes place in natural language, it is difficult to categorise how the use of bug finding and patching tools are valued in the process. Nevertheless, we can use the mailing list to study the frequency at which developers specifically mention bug finding tools when a patch is first submitted. Then, we can correlate this frequency on a monthly basis with the corresponding statistics on accepted DLH patches related to the specific tools.



(a) Data on checkpatch-related (DLH) patches.

(b) Evolution of the Gap.

(c) Data on coccinelle-related (HMG) patches.

(d) Evolution of the Gap.

Figure 3.7: # of Patches submitted / discussed / accepted.

We have crawled all emails archived in the Linux Kernel Mailing List (LKML) using Scrapy[14]. We use heuristics to differentiate message replies from original mail content: we consider lines starting with '>' as part of a previous conversation. Finally, we naively search for the tool name reference in the message text. In total, we crawled[15] 1,601,606 original email messages and 885,814 reply messages. As examples, we provide in Figures 3.7a and 3.7c the distributions per month of the number of patches that were submitted through LKLM mentioning *checkpatch* or coccinelle respectively, as well as the number of maintainer replies referencing those tools, and the number of related commits accepted into the mainline git tree. To ease observation, we compute in Figures 3.7b and 3.7d the integration gap as a percentage between the number of patches submitted to LKML and the number of patches that are eventually integrated. We draw the slope of the evolution of this gap over time. While checkpatch presents roughly the same gap, the gap is clearly reducing for coccinelle. We have computed the slope for the different sets of tool-supported patches and checked that it was negative for 3 out of 4 of the tools[16]: the gap is thus closing over time for most tool-supported processes.

Tool-supported patches (DLH and HMG alike) have been overall accepted at an increasing rate by Linux developers. Integration of such patches by maintainers remains, however, slower than that of traditional H patches.

[14] `https://scrapy.org/`, a framework for deploying and running spiders

[15] 7,510 entries were empty messages and were thus dropped out.

[16] We considered only tools associated to at least 50 patches.

3.4.3 Profile of Patch Authors (RQ2)

We investigate the speciality and commitment of developers who rely on patch application and bug finding tools to construct patches.

Speciality is defined as a metric for characterising the extent to which a developer is focused on a specific subsystem. We compute it as the percentage of patches, among all her/his patches, which a developer contributes to a specific subsystem. Thus, the *speciality* is measured with respect to each Linux code directory. We then draw, in Figure 3.8, the distributions of speciality metric values of developers for the different types of patches: e.g., for an automated patch applied to a file in a subsystem, we consider the commit author speciality w.r.t that subsystem.

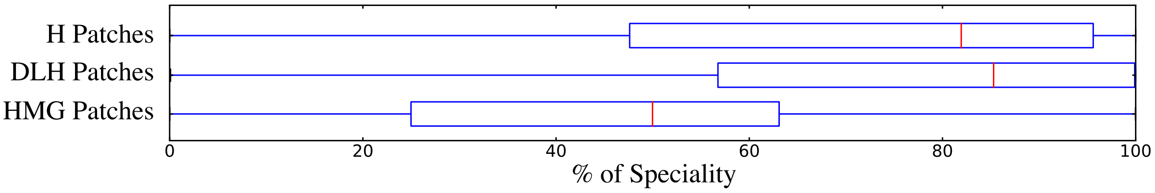


Figure 3.8: Speciality of developers `Vs.` Patch types.

H patches are mostly provided by specialised developers. This may imply that the developers focus on implementing specific functionalities over time. Similarly, DLH patches appear to be mostly applied by specialised developers (even slightly more specialised than those who made H patches). This finding is in line with the requirements for developers to be aware of the idiosyncrasies of the programming of a particular subsystem to validate the warnings of bug detection tools and sift through various false positives to produce patches that are eventually accepted by maintainers. HMG patches, on the other hand, are performed by developers on subsystem code which they are not known to be specialised on.

To measure developer *commitment*, we follow the approach of Palix et al. [180] and compute, for each developer, the product of (1) the number of patches (H, DLH or HMG) that have been integrated into Linux and (2) the number of days between the first patch and the last patch. This metric favours both developers who have contributed many patches over a short period of time and developers who have contributed fewer patches over a longer period of time: e.g., a developer who gets 10 commits integrated during one year, will have the same degree of commitment as another developer who gets 40 commits integrated in 3 months.

Developer *commitment* is studied here as an approximation of developer expertise, since the more a developer works on the Linux project or with a tool, the more expertise the developer may be assumed to acquire (on the Linux project and/or with the use of the tool). Figure 3.9 shows the distribution of commitment scores of developers for the different types of patches.

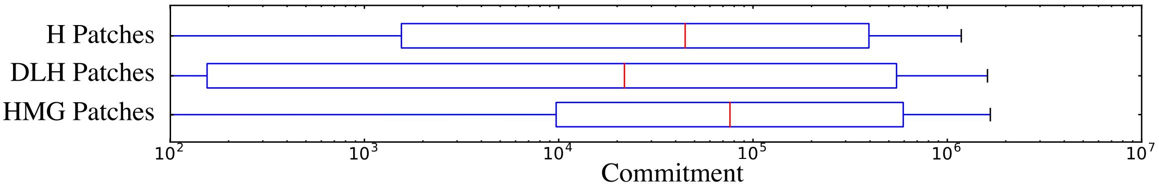


Figure 3.9: Commitment of developers `Vs.` Patch types.

DLH patches are shown to be produced by developers with a more varying degree of commitment (greater standard deviation). The median value of commitment is further lower than the median commitment for HMG patches. Finally, overall, the distributions of commitment values of developers indicate that H patch authors present lesser commitment than HMG patch authors.

We then use Spearman's ρ [215] to measure the degree of correlation between the commitment of developers and the number of tool-supported patches that they submit. We focus on specialised[17] developers of two very different kinds of code: mature file system (*fs*) code and early-development (*staging*) code. The correlation is then revealed to be higher ($\rho = 0.42$) for staging than for fs ($\rho = 0.11$). We also note that 64% of developers committing code in staging stick to this part of the code for over half of their contributions. Finally, developers specialised in *kernel* have never relied on tool support to produce a patch.

> Bug detection tools are generally used by developers with (to some extent) knowledge of the code. Patch application tools, on the other hand, enable developers to remain committed to contributing patches to the code base.

3.4.4 Stability of Patches (RQ3)

Although patches are carefully validated before they are integrated to the mainline code base, a patch might be simply incorrect, and thus the relevant code may require further changes or the patch may simply be reverted. However, it is challenging to precisely detect and resolve such a change in recently patched code hunks. Even this requires heuristics that may prove to be error-prone. Thus, in this study, we focus on commits whose reverting is explicit.

It is common for software developers to cancel patches that they hastily committed to the code base. The `git revert` command is an excellent means for developers to roll back their commits. However, given the hierarchical organisation in Linux, when a patch has reached the mainline, a simple revert (using git commands) is uncommon. The submitting developer (or another one) must write another patch explaining the need to revert. This patch again goes through the process to be accepted in the mainline. In this setting, the revert of a commit is likely strongly justified. We search for commits that are reverted by looking at commit messages where we have seen a pattern of the form "`revert <hash>`"[18].

We have found that 2.81% of H-patch commits have been later reverted. In contrast, only 0.27% and 0.32% respectively of DLH and HMG patch commits have been reverted. Figure 3.10 further provides the distributions of delays in reverting commits.

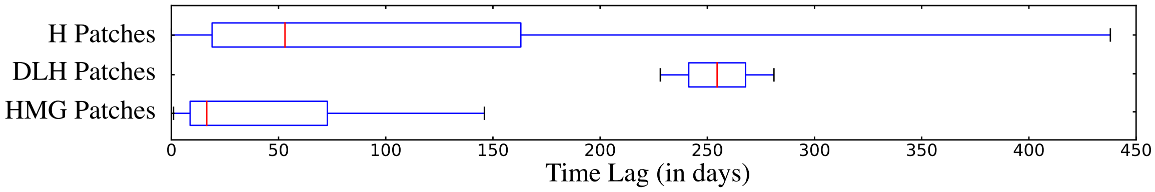


Figure 3.10: Time lag between patch integration and reverting.

H-patches revert delay distribution is the most spread. On average (median), a DLH patch, when it is reverted, will be so after 250 days (8 months). On the other hand, HMG patches will be reverted in less than a month (20 days). The median delay for revert is of 60 days for H patches.

> Tool-supported patches are generally stable. However, while patches fixing tool warnings may be found inadequate long after their integration, issues with patches generated based on fix patterns appear to be discovered quickly.

[17] speciality metric value greater than 50%

[18] We use: `git show '+sha+' | grep -E -i "revert .[0-9a-f]5+ | commit .[0-9a-f]5+ | [0-9a-f]{40}$*`

3.4.5 Bug Kinds (RQ4)

We study bug kinds in two dimensions: the spread of buggy code and the complexity of the bugs. We investigate the locality of patches as an approximation of the spread of buggy code, and the change operations at the level of Abstract syntax tree nodes modifications to approximate the complexity of bugs.

3.4.5.1 Locality of patches

The locality of patches is a key dimension for characterising patches. Patch size has been measured in the literature [24, 180] in terms of the number of code locations that it involves, while several state-of-the-art automated repair approaches mostly focus on single/limited code changes to fix software. The Linux project is a particularly adequate study subject for this comparison since developers are often reminded that they must "solve a single problem per patch"[19]: fix operations are then generally separated from cosmetic changes.

A bug fix patch may involve changes across files. Figure 3.11 shows that most fixes are localized to a single file independently of the way they are constructed.

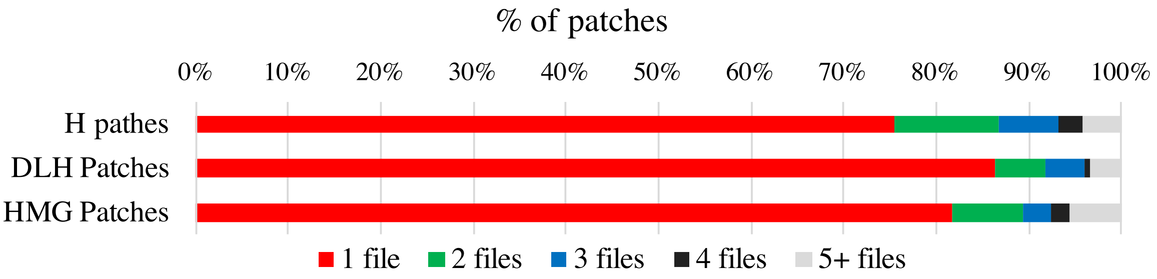


Figure 3.11: Distribution of patch sizes in terms of files.

DLH patches appear to be the more local, while more than 20% of H patches implement simultaneous changes in at least two files. Interestingly, we note that HMG patches include the largest proportion of patches (5.6%) that simultaneously change 5 files or more. Such patches are generated to fix pervasive bugs such as the wrong usage of an API, or to implement a collateral evolution.

We further investigate the locality of patches in terms of the number of code hunks (i.e., a contiguous group of code lines[20]) that are changed by a patch. Indeed, code files can be large, and a patch may variably spread changes inside the file, which, to some extent, may represent a degree of complexity of the fix. Figure 3.12 shows that H patches are more likely to involve several hunks of code than HMG and DLH patches.

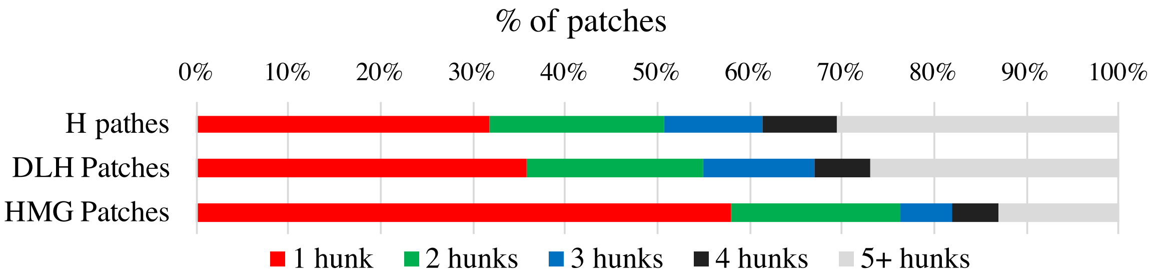


Figure 3.12: Distribution of patch sizes in terms of hunks.

[19] see `Documentation/SubmittingPatches`

[20] `https://www.gnu.org/software/diffutils/manual/html_node/Hunks.html`

Our observations on patch sizes suggest that developers, with or without bug finding tools, must correlate data and code statements across different code blocks to repair programs.

Finally, we compute the locality of the patches in terms of the number of lines that are affected by the changes. Such a study is relevant for estimating the proportions of isolated change (i.e., single-line changes) that fix bugs in the three scenarios of repairs. Figure 3.13 reveals that the large majority of patches that are manually crafted as responses to bug reports change several lines, with almost 70% patches impacting at least 5 lines. On the other hand, over 40% of HMG patches impact only at most two lines of code.

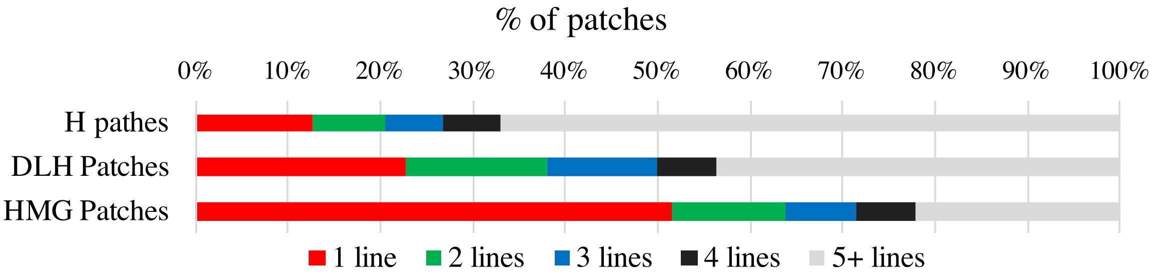


Figure 3.13: Distribution of patch sizes in terms of lines.

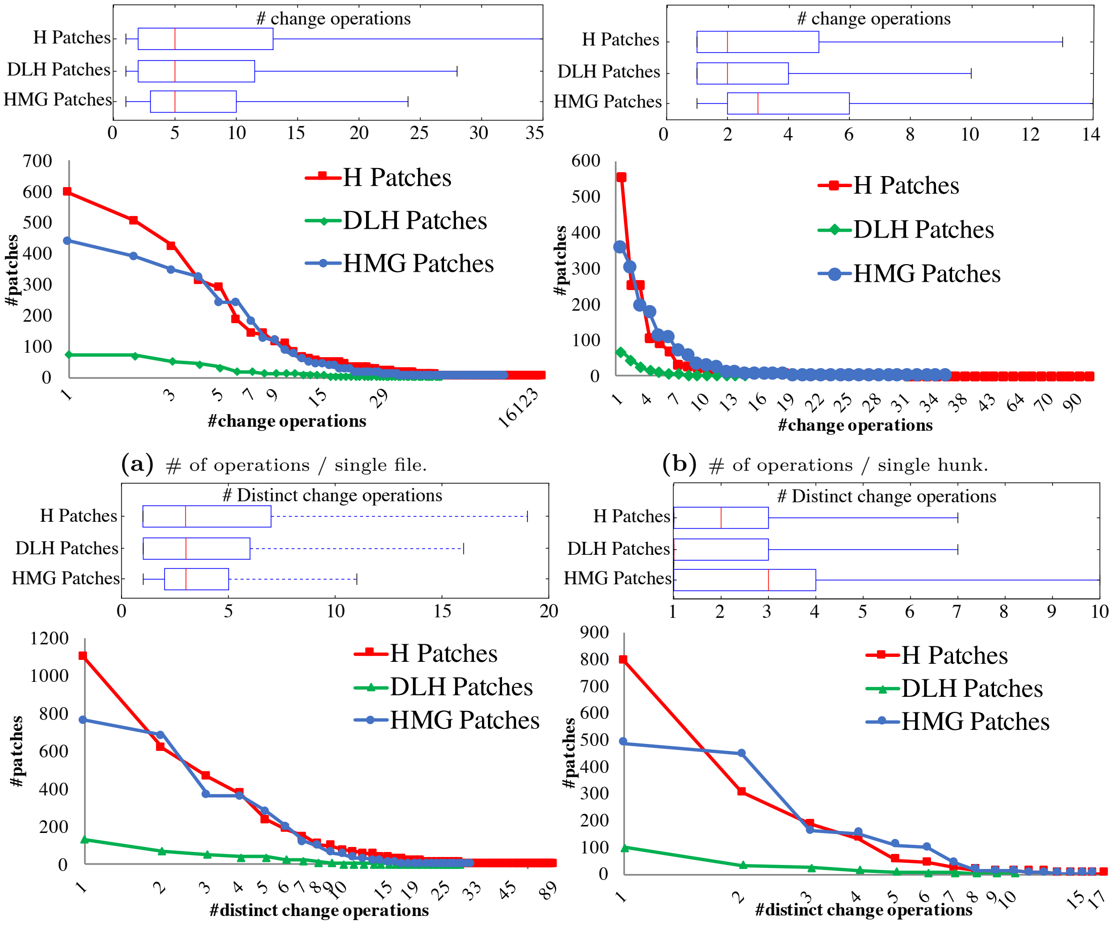
3.4.5.2 Change operations in patches

In general, line-based diff tools, such as the GNU Diff, are limited in the expression of the kinds of changes that can be identified since they consider only adds and removes, but no moves and updates [178]. Thus, to investigate change operations performed by patches, we rely on approaches that compute modifications based on abstract syntax trees (AST) [90]. Such approaches produce fine-grained results at the level of individual nodes. For this study, we consider an extended version of the open-source GumTree [52] with support for the C language [178]. This tool specifically takes into account additions, deletions, updates and moves of individual tree nodes, and has the goal of producing results that are easier for users to understand than those of GNU Diff.

The output of GumTree is an edit script enumerating a sequence of operations that must be carried out on an AST tree to yield the other tree. To that end, GumTree implements a mapping algorithm between the nodes in two abstract syntax trees. This algorithm is inspired by the way developers manually look at changes between two files, first searching for the largest unmodified chunks of code (i.e., isomorphic subtrees) and then identifying modifications (i.e., given two mapped nodes, find descendants that share a large percentage of common mappings, and so on). Given those mappings, GumTree leverages an optimal and quadratic algorithm [33] to compute the edit script. More details on the algorithm can be found in the original articles [33, 52].

For simplicity, we express change operations in their abstract form as a triplet "*scope/element:action*" where *scope* represents the type of node (e.g., the program, an `If` block, a compound block, a generic list, an identifier, etc.) where the change occurs, *element* represents the element (e.g., an expression, a declaration, a generic string, a compound block, an if block, etc.) that is changed and *action* represents the move/update/add/delete operators that are used. This abstract representation indeed does not take into account any variable names and functions involved (and available in the output of GumTree). Figure 3.16 shows a patch example for a change operation where a new `If` block code is inserted.

Figure 3.14 illustrates the distributions of the number of operations that are performed in a patch. To limit the bias of changes that are identically performed in several files (e.g., Coccinelle collateral evolutions), we focus on patches that touch a single file, then on patches that are limited to a single hunk. All distributions are long-tail, revealing that most patches apply very few operations in terms



(a) # of operations / single file.

(b) # of operations / single hunk.

(c) # of distinct operations / single file.

(d) # of distinct operations / single hunk.

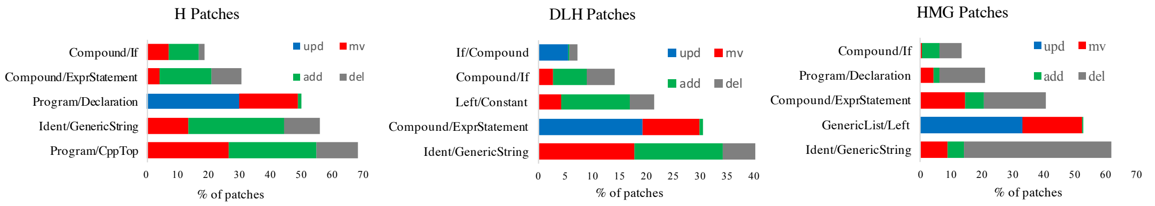
Figure 3.14: Distribution of change operations (Total # of operations & # of distinct operations in patches).

of number and variety. While the three processes have similar average (median) values of change operations performed on a file, HMG patches appear to implement changes with a consistent number of operations (limited standard deviation). On the other hand, when we consider change operations at the hunk level, DLH patches apply fewer operations than HMG patches[21].

Figure 3.15 summarises the top-5 change operations that are recurrently implemented by patches constructed in the different processes considered in our study. Changes performed appear to be specific for each process. For example, while `Ident/GenericString` and `Compound/If`-related change operations occur in most patches, they do not display the same proportions in terms of additions, moves, updates and deletions.

> Overall, patches, following their construction process, differ in terms of size (i.e., the spread of the buggy code that they repair) and in the nature of change operations that they implement (i.e., the complexity of the bug).

[21] We have checked with MWW tests that the difference is statistically significant.



(a) H patches. **(b)** DLH patches. **(c)** HMG patches.

Figure 3.15: Top-5 change operations appearing at least once in a patch from the three processes.

```
diff --git a/drivers/gpu/drm/i915/intel_display.c b/.../drm/i915/intel_display.c
index 6e0d828..182f849 100644

--- a/drivers/gpu/drm/i915/intel_display.c
+++ b/drivers/gpu/drm/i915/intel_display.c
@@ -13351,6 +13351,9 @@ int intel_atomic_prepare_commit(struct drm_device *dev,
        for_each_crtc_in_state(state, crtc, crtc_state, i) {
+               if (state->legacy_cursor_update)
+                       continue;
+
                ret = intel_crtc_wait_for_pending_flips(crtc);
```

Figure 3.16: Example of `Compound/If:add` – Add an `If` block.

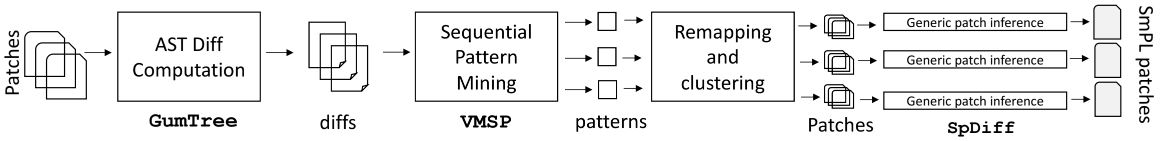


Figure 3.17: Searching for redundancies among patches that fix warnings of bug finding tools (i.e., DLH patches).

3.5 Discussions

We discuss the implications of our findings for the software engineering research community, in particular, the automated research field, and enumerate the threats to validity that this study carries.

3.5.1 Implications

As the field of automated repair is getting mature, the community has started to reflect (i) on whether to build human-acceptable or readable patches [88,162], (ii) on the suitability of automated repair fixes [210], (iii) on the relevance of patches produced by repair tools [275]. Our work continues this reflection from the perspective of the acceptance of tool-support in patch construction. We further acknowledge that HMG patches considered in this study are not constructed in the same spirit as in automated repair: indeed, automated repair approaches make no a-priori assumption on what and where the fault is, while tools such as Coccinelle [28] produce patches based on fix patterns that match buggy code locations. Nevertheless, given the lack of integration of automated repair in a real-world development process, we claim that investigating Linux patch cases can offer insights which can be leveraged by the research community to understand how the developer community can accept tool-supported patches, and the automation of what kind of fixes can be readily accepted in the community.

On manual Vs. tool-supported patches. As illustrated in Section 3.4.1, tool-supported patch construction is becoming frequently and widely used in the Linux Kernel development. In particular, HMG patches account for a larger portion of recent program changes than H patches. This suggests that both (1) developers gradually accept to use patch application tools such as Coccinelle [28] since they are effective to automatically change similar code fragments and (2) there are many (micro) code clones [226] in the code base. Regarding spatial distribution, DLH and HMG patches are committed to 'staging' (22-47%) while H patches in 'staging' account for only 1%. This may indicate that experimental features have more opportunities for tools to help write bug fixing patches. It implies indeed that, for early development code, the community almost exclusively relies upon tools to solve common bugs (e.g., in relation with programming rules, styles, code hardening, etc.) by novice programmers (i.e., not necessarily specialised in kernel code), before expert developers can take over. Thus, reliable automated repair techniques could be beneficial in a production development chain as debugging aids. This finding comforts the human study recently conducted by Tao et al. [221] which suggested that automated repair tools can significantly help debugging tasks.

On the delay in patch acceptance. We have observed a delay in the acceptance of tool-supported patches by maintainers. However, given the differences in change operations with fully manual patches, it is likely the case that tool-supported patches are fixing less severe bugs, which makes their integration a less crucial issue for maintainers.

Furthermore, negative percentages in evolution gap between submission and acceptance (cf. Figure 3.7) suggests that there are many HMG patches that are integrated into the mainline code base without being discussed by maintainers. This finding implies that once the fix pattern has been validated, patches appear to be accepted systematically.

On the nature of bugs being fixed. The study of patch locality shows results that are in line with a previous study [275] which revealed that most fix patches only change a single file. Nevertheless, we have found that, in practice, even tool-supported patches, in a large majority, modify several lines to fix warnings by bug detection tools (which, by the way, generally flag a single line in the code). Although patch size does not, by any means, imply ease of realisation, our results suggest that there are considerable numbers of repair targets and shapes that automated repair should aim for.

It is also noteworthy that the spread of change operations over several files may carry different implications for the patch construction processes. For example, while a coccinelle patch may be applying the same change pattern over several files to fix an API function usage, a human patch modifying several files may actually carry data and behaviour dependencies among the changes.

3.5.2 Exploiting Patch Redundancies

A large body of the literature on program repair has discussed findings on the repetitiveness/redundancy of code changes in real-world software development [13, 173]. Unfortunately, such findings are not readily actionable in the context of automated repair since they do not come with insights on how such redundant patches will be leveraged in practice. Indeed, although it is possible to abstract redundant patches to recommend bug fix actions [20], only a few research directions manage to contextualise them, to some extent, for repair scenarios [139]. Actually, researchers discuss such redundancies for enriching the repair space with change operations that are more likely to be appropriate fix operations.

With this study, we see concrete opportunities for exploiting patch redundancies for systematically building patches and applying (or recommending) them to a specific identified and localised buggy piece of code. Indeed, bug detection tools, which are used by various developers who then craft fixes based on specific warnings, and patch application tools, which are based on fix patterns, can be leveraged in an automated repair chain. The former will be used in the bug detection and localisation steps while the latter will focus on building concrete patches based on patterns found in a database of human fixes created to address warnings by bug detection tools.

To demonstrate the feasibility of this research direction, we have conducted a study for searching redundancies in patches constructed following warnings by bug detection tools, and investigating the possibility of producing a generic patch which could have been used to derive these concrete patches. Nevertheless, although generic patch inference has been a very fertile research direction in the past [9, 10, 157, 159], we have experimented available tool supports and found that they do not scale in practice. We have thus devised a process to split the set of patches into clusters, each containing patches presenting similar change operations. Figure 3.17 depicts the overall process. Based on GumTree sequences of change operations, we rely on a sequential pattern mining tool to extract maximal sequential patterns. We use a fast implementation of VMSP [57] to find recurrent change patterns at the level of the abstract change operations expressed in Section 3.4.5. Then, we build clusters of patches based on the elicited patterns, and leverage SpDiff [9] to attempt the inference of a unique SmPL patch which could instantiate the common redundant concrete repair actions performed in the patches.

With this process, starting with a set of 571 DLH patches, we were able to build 37 clusters based on change operations patterns. Among the clusters, 10 led to the generation of a common generic patch. We then manually investigated the commit messages associated with the patches in clusters that produced a generic patch, and found that they indeed largely dealt with the same bug type. This final check confirms, to some extent, the potential to collect fix patterns from human repair processes to build an automated repair chain leveraging bug detection tools.

3.5.3 Threats to Validity

We have identified the following threats to validity to our study:

External validity – We focus on Linux only. It is, however, one of the largest development project, one of the most diverse in terms of developer population, with a significant history for observing trends, and implementing strict patch submission guidelines that try to systematise the tracking of change information. To the best of our knowledge, Linux is the best candidate for observing various patch construction processes, as it encourages the use of tools for bug detection and patching.
Construct validity – We rely on a number of heuristics to collect and process our datasets. We have nevertheless, by design, chosen to be conservative in the way we collect patches in each process with the objective of having reliable and distinctive sets for each process, to further enable replication.
Internal validity – The metrics that we leverage to elicit the differences among the different processes may lead to biased results. However, those metrics were also used in the literature.

3.6 Related Work

3.6.1 Program Repair

3.6.1.1 Studies on human-generated patches

Studies on patches, generated by human developers, focus on investigating existing patches fully written by developers (i.e., H patches) rather than devising a new technique. Pan et al. explored syntactic bug fix patterns in seven Java projects [181]. This study extracted 27 bug fix patterns. Martinez and Monperrus identified common program repair actions (patterns) [151], and Zhong and Su reported statistics on 9,000 real bug fixing patches collected from Java open source projects [275]. These studies examined features of real bug fixes against whether automated repair techniques can be applied to fix those bugs. In addition, Barr et al. formulated a hypothesis called "*plastic surgery hypothesis*" [13]. They studied how many changes can be graftable by using snippets that can be found in the same code base where the changes are made.

3.6.1.2 Studies on tool-aided patches

As discussed in Sections 3.1 and 3.3, generating tool-aided patches indicates that developers create program patches with an aid of tools, rather than generating patches from scratch. Tao et al. supposed that automated repair tools can provide aids to debugging tasks [221]. They adopted PAR [88] as a patch recommendation tool and gave patches generated by the tool to experiment participants. The findings include that automatically generated patches can significantly help debugging tasks. MintHint [81] is a semi-automatic repair technique, which can help developer find correct patches. This technique does a statistical correlation analysis to locate program expressions likely to perform repaired program executions.

3.6.1.3 Automated patch generation

Generating patches with automated tools implies minimising a developer's effort in debugging. It often indicates that fully automated procedures including fault localisation, code modification, and patch verification. Recent endeavours achieved impressive progress as follows.

Weimer et al. [237] proposed GenProg, an automatic patch generation technique based on genetic programming [101]. This technique randomly mutates buggy statements to generate several different program variants that are potential patch candidates. In 2012, the authors extended their previous work by adding a new mutation operation, replacement and removing the switch operation [117]. SemFix [172] leverages program synthesis to generate patches. The technique assumes that buggy predicates are an unknown function to be synthesised. The technique is successful for several bugs, but it is only applicable to "one-line bug", in which only one predicate is buggy. DirectFix [155] and Angelix [156] extended Semfix so that it can generate patches for bugs in larger and complex (w.r.t the search space) programs in a simpler way. PAR [88] automatically generates patches by using fix patterns learned from human-written patches. This technique is inspired by the fact that patches are redundant.

3.6.2 Patch Acceptability

Fry et al. conducted a human study to indirectly measure the quality of patches generated by GenProg by measuring patch maintainability [59]. They presented patches to participants and asked maintainability related questions developed by Sillito et al. [207]. They found that machine-generated patches [117] with machine-generated documents [29] are comparable to human-written patches in terms of maintainability. PAR [88] is presented to deal with nonsensical patches. The approach generates patches based on fix patterns, which are learned from human-written patches. The fix patterns generalise common repair actions from more than 60,000 real bug fixes enabling PAR to avoid generating nonsensical patches.

3.6.3 Program Matching and Transformation

SYDIT [157] automatically extracts an edit script from a program change. In its scenario, a user must specify the program change to extract the edit script from. Coccinelle [28], on the other hand, directly lets the user specify the edit script in a user-friendly language, and performs the transformation by matching the change pattern with code context. It has been used in several debugging tasks in the literature [19, 20, 22, 23, 179]. LASE [159] differs from SYDIT as it can generate a generalised edit script based on multiple changes of Java programs. Another approach in this direction is SpDiff [9, 10] supports the extraction of a subset of common changes (i.e., SmPL patches that are fed to Coccinelle) from several concrete patches.

3.7 Summary

We have studied the impact of tool support in patch construction, leveraging real-world patching processes in the Linux kernel development project. We investigated the acceptance of tool-supported patches in the development chain as well as the differences that may exist in the kinds of bugs that such patches fix in comparison with traditional all handwritten patches. The result of our study shows that tool-supported patches are increasingly adopted by the developer community while manually-written patches are accepted more quickly. Patch application tools enable developers to remain committed to contributing patches to the code base. Our findings also include that, in actual development processes, patches generally implement several change operations spread over the code, even for patches fixing warnings by bug detection tools. Finally, this study has shown that there is an opportunity to directly leverage the output of bug detection tools to readily generate patches that are appropriate for fixing the problem, and that is consistent with manually-written patches.

Overall, we show that in the Linux ecosystem, bug detection and patch application tools are already heavily used to unburden developers, and already enable relatively complex repair schema, contrasting with a number of repair approaches in the state-of-the-art literature of automated repair.

Building on insights this empirical study, in the remainder of this dissertation, we mainly focus on following three research axes towards devising practical automated repair approaches: First, **mining software repositories** towards understanding code change properties that could be valuable to guide program repair. Second, **analysing communication channels** in software development in order to assess to what extent they could be relevant in a real-world program repair scenario. Third, **exploring generic concepts of patching** in the literature for establishing a common foundation for program repair pipelines that can be integrated with industrial settings.

4 Mining Software Repositories

Patching is a common activity in software development. It is generally performed on a source code base to address bugs or add new functionalities. In this context, given the recurrence of bugs across projects, the associated similar patches can be leveraged to extract generic fix actions. While the literature includes various approaches leveraging similarity among patches to guide program repair, these approaches often do not yield fix patterns that are tractable and reusable as actionable input to APR systems.

We propose a systematic and automated approach to mining relevant and actionable fix patterns based on an iterative clustering strategy applied to atomic changes within patches. The goal of `FixMiner` is thus to infer separate and reusable fix patterns that can be leveraged in other patch generation systems. Our technique, `FixMiner`, leverages `Rich Edit Script` which is a specialised tree structure of the edit scripts that captures the AST-level context of the code changes. `FixMiner` uses different tree representations of `Rich Edit Scripts` for each round of clustering to identify similar changes. These are abstract syntax trees, edit actions trees, and code context trees.

This chapter is based on the work published in the following research paper:

- A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020

Contents

4.1 Overview

Code change patterns have various uses in the software engineering domain. They are notably used for labelling changes [181], triaging developer commits [225] or predicting changes [265]. In recent years, fix patterns have been heavily leveraged in the software maintenance community, notably for building patch generation systems, which now attract growing interest in the literature [163]. Automated Program Repair (APR) has indeed gained incredible momentum, and various approaches [35, 39, 72, 76, 84, 88, 110, 111, 114, 117, 131, 132, 137, 138, 140, 155, 172, 238, 241, 255, 257] have been proposed, aiming at reducing manual debugging efforts through automatically generating patches. A common and reliable strategy in automatic program repair is to generate concrete patches based on fix patterns [88] (also referred to as fix templates [135] or program transformation schemas [72]). Several APR systems [48, 72, 88, 131, 132, 135, 153, 200] in the literature implement this strategy by using **diverse sets of fix patterns** obtained either via manual generation or automatic mining of bug fix datasets.

In PAR [88], the authors mined fix patterns by inspecting 60,000 developer patches manually. Similarly, for Relifix [219], a manual inspection of 73 real software regression bug fixes is performed to infer fix patterns. Manual mining is, however tedious, error-prone, and cannot scale. Thus, in order to overcome the limitations of manual pattern inference, several research groups have initiated studies towards automatically inferring bug fix patterns. With Genesis [137], Long *et al.* proposed to infer code transforms for patch generation automatically. Genesis infers 108 code transforms, from a space of 577 sampled transforms, with specific code contexts. However, this work limits the search space to previously successful patches from only three classes of defects of Java programs: null pointer, out of bounds, and class cast related defects.

Liu and Zhong [135] proposed SOFix to explore fix patterns for Java programs from Q&A posts in Stack Overflow, which mines patterns based on GumTree [52] edit scripts, and builds different categories based on repair pattern isomorphism. SOFix then mines a repair pattern from each category. However, the authors note that most of the categories are redundant or even irrelevant, mainly due to two major issues: (1) a considerable portion of code samples are designed for purposes other than repairing bugs; (2) since the underlying GumTree tool relies on structural positions to extract modifications, these "modifications do not present the desirable semantic mappings". They relied on heuristics for manually filtering categories (e.g., categories that contain several modifications), and then after SOFIX mines repair patterns they have to manually select useful ones (e.g., merging some repair patterns due to their similar semantics).

Liu et al. [129] and Rolim et al. [196] proposed to mine fix patterns from static analysis violations from FindBugs and PMD, respectively. Both approaches leverage a similar methodology in the inference process. Rolim et al. [196] rely on the distance among edit scripts: edit scripts with low distances among them are grouped together according to a defined similarity threshold. Liu et al. [129], on the other hand, leverage deep learning to learn features of edit scripts, to find clusters of similar edit scripts. Eventually, both works do not consider code context in their edit scripts and manually derive the fix patterns from the clusters of similar edit scripts of patches.

In another vein, CapGen [241] and SimFix [76] propose to use the frequency of code change actions. The former uses it to drive patch selection, while the latter uses it in computing donor code similarity for patch prioritisation. In both cases, however, the notion of patterns is not an actionable artefact, but rather a supplementary information that guides their patch generation system. Although we concurrently[1] share with SimFix and CapGen the idea of adding more contextual information for patch generation, our objective is to infer actionable fix patterns that are tractable and reusable as input to other APR systems.

[1] The initial version of this work was written concurrently to SimFix and CapGen.

Table 4.1 presents an overview of different automated mining strategies implemented in the literature to obtain diverse sets of fix patterns. Some of the strategies are directly presented as part of APR systems, while others are independent approaches. We characterise the different strategies by considering the diff representation format, the use of contextual information, the tractability of patterns (i.e., what extent they are separate and reusable components in patch generation systems), and the scope of mining (i.e., whether the scope is limited to specific code changes). Overall, although the literature approaches can come handy for discovering diverse sets of fix patterns, the reality is that the intractability of the fix patterns and the generalisability of the mining strategies remains a challenge for deriving relevant patterns for program repair.

Table 4.1: Comparison of fix pattern mining techniques in the literature.

	Genesis [137]	SOFix [135]	Liu et al. [129]	Rolim et al. [196]	CapGen [241]	SimFix [76]	FixMiner
Diff notation	Transform	Edit Script	Edit Script	Edit Script	Edit Script	Edit Script	Edit Script
Scope	Three defect classes	Any bug type	Static analysis violations	Static analysis violations	Any bug type	Insert and update changes only	Any bug type
Context information	✗	✗	✗	✗	✓	✓	✓
Tractability of Patterns*	Medium	High	High	High	Low	Low	High

[*] **High:** Patterns are part of output and reusable as input to APR systems
Medium: Patterns are not readily usable
Low: Patterns are not separate or available as output.

We propose to investigate the feasibility of mining relevant fix patterns that can be easily integrated into an automated pattern-based program repair system. To that end, we propose an iterative and three-fold clustering strategy, `FixMiner`, to discover relevant fix patterns automatically from atomic changes within real-world developer fixes. `FixMiner` is a pattern mining approach to produce fix patterns for program repair systems. We present the concept of `Rich Edit Script`, which is a specialised tree data structure of the edit scripts that captures the AST-level context of code changes. To infer patterns, `FixMiner` leverages identical trees, which are computed based on the following information encoded in `Rich Edit Scripts` for each round of the iteration: abstract syntax tree, edit actions tree, and code context tree.

Contribution. We propose the `FixMiner` pattern mining tool as a separate and reusable component that can be leveraged in other patch generation systems.

Our contributions are:

- We present the architecture of a pattern inference system, `FixMiner`, which builds on a three-fold clustering strategy where we iteratively discover similar changes based on different tree representations encoding contexts, change operations and code tokens.
- We assess the capability of `FixMiner` to discover patterns by mining fix patterns among 11 416 patches addressing user-reported bugs in 43 open source projects. We further relate the discovered patterns to those that can be found in a dataset used by the program repair community [78]. We assess the compatibility of `FixMiner` patterns with patterns in the literature.
- Finally, we investigate the relevance of the mined fix patterns by embedding them as part of an Automated Program Repair system. Our experimental results on the Defects4J benchmark show that our mined patterns are effective for fixing 26 bugs. We find that the `FixMiner` patterns are relevant as they lead to generating plausible patches that are mostly correct.

4.1.1 Motivation

Mining, enumerating and understanding code changes have been a key challenge of software maintenance in recent years. Ten years ago, Pan et al. contributed with a manually-compiled catalogue of 27 code change patterns related to bug fixing [181]. Such "bug fix patterns" however are generic patterns (e.g., IF-RMV: removal of an If Predicate) which represent the type of changes that are often fixing bugs. More recently, thanks to the availability of new AST differencing tools, researchers have proposed to automatically mine change patterns [124, 149, 175, 176]. Such patterns have been mostly leveraged for analysing and towards understanding characteristics of bug fixes. In practice, however, the inferred patterns may turn out to be irrelevant and intractable.

We argue, however, that mining fix patterns can help for guiding mutation operations for patch generation. In this case, there is a need to mine truly recurrent change patterns to which repair semantics can be attached, and to provide accurate, fine-grained patterns that can be actionable in practice, i.e., separate and reusable as inputs to other processes. Our intuition is that relevant patterns cannot be mined globally since bug fixes in the wild are subject to noisy details due to tangled changes [68]. There is thus a need to break patches into atomic units (contiguous code lines forming a hunk) and reason about the recurrences of the code changes among them. To mine changes, we propose to rely on the edit script format, which provides a fine-grained representation of code changes, where different layers of information are included:

- the context, i.e., AST node type of the code element being changed (e.g., a modifier in declaration statements, should not be generalised to other types of statements);
- the change operation (e.g., a "remove then add" sequence should not be confused with "add then remove" as it may have a distinct meaning in a hierarchical model such as the AST);
- and code tokens (e.g., changing calls to "*Log.warn*" should not be confused to any other API method).

Our idea is to find patterns within the contexts iteratively, and patterns of change operations for each context, and patterns of recurrently affected literals in these operations.

We now provide background information for understanding the execution as well as the information processed by `FixMiner`.

4.2 Background

4.2.1 Abstract Syntax Tree

Code representation is an essential step in the analysis and verification of programs. Abstract syntax trees (ASTs), which are generally produced for program analysis and transformations, are data structures that provide an efficient form of representing program structures to reason about syntax and even semantics. An AST indeed represents all of the syntactical elements of the programming language and focuses on the rules rather than elements like braces or semicolons that terminate statements in some popular languages like Java or C. The AST is a hierarchical representation where the elements of each programming statement are broken down recursively into their parts. Each node in the tree thus denotes a construct occurring in the programming language.

Formally, let t be an AST and N be a set of AST nodes in t. An AST t has a root that is a node referred to as $root(t) \in N$. Each node $n \in N$ (and $n \neq root(t)$) has a parent denoted as $parent(n) = p \in N$. Note that there is no parent node of $root(t)$. Furthermore, each node n has a set of child nodes (denoted as $children(n) \subset N$). A label l (i.e., AST node type) is assigned to each node from a given alphabet L ($label(n) = l \in L$). Finally, each node has a string value v ($token(n) = v$ where $n \in N$ and v is an arbitrary string) representing the corresponding raw code token. Consider the AST representation in Figure 4.2 of the Java code in Figure 4.1. We note that the illustrated AST has nodes with labels matching structural elements of the Java language (e.g., `MethodDeclaration`, `IfStatement` or `StringLiteral`) and can be associated with values representing the raw tokens in the code (e.g., A node labelled `StringLiteral` from our AST is associated to value "Hi!").

```
public class Helloworld {
    public String hello(int i) {
        if (i == 0) return "Hi!";
    }
}
```

Figure 4.1: Example Java class.

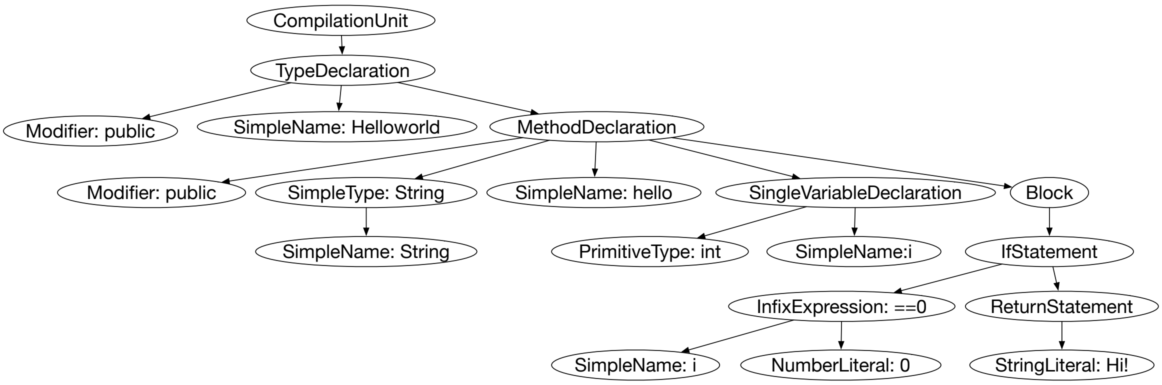


Figure 4.2: AST representation of the `Helloworld` class.

4.2.2 Code Differencing

Differencing two versions of a program is the key pre-processing step of all studies on software evolution. The evolved parts must be captured in a way that makes it easy for developers to understand or analyse the changes. Developers generally deal well with text-based differencing tools, such as the GNU Diff represents changes as addition and removal of source code lines, as shown in Figure 4.3. The main issue with this text-based differencing is that it does not provide a fine-grained representation of the change (i.e., `StringLiteral Replacement`) and thus it is poorly suited for systematically analysing the changes.

```
--- Helloworld_v1.java  2018-04-24 10:40:19.000000000 +0200
+++ Helloworld_v2.java  2018-04-24 11:43:24.000000000 +0200
@@ -1,5 +1,5 @@
 public class Helloworld {
     public String hello(int i) {
-        if (i == 0) return "Hi!";
+        if (i == 0) return "Morning!";
     }
 }
```

Figure 4.3: GNU diff format.

To address the challenges of code differencing, recent algorithms have been proposed based on tree structures (such as the AST). ChangeDistiller and GumTree are examples of such algorithms which produce *edit scripts* that detail the operations to be performed on the nodes of a given AST (as formalised in Section 4.2.1) to yield another AST corresponding to the new version of the code. In particular, in this work, we build on GumTree's core algorithms for preparing an edit script. An edit script is a sequence of edit actions describing the following code change actions:

- `UPD` where an $upd(n, v)$ action transforms the AST by replacing the old value of an AST node n with the new value v.
- `INS` where an $ins(n, n_p, i, l, v)$ action inserts a new node n with v as value and l as label. If the parent n_p is specified, n is inserted as the i^{th} child of n_p, otherwise n is the root node.
- `DEL` where a $del(n)$ action removes the leaf node n from the tree.

- MOV where a $mov(n, n_p, i)$ action moves the subtree having node n as root to make it the i^{th} child of a parent node n_p.

An edit action embeds information about the node (i.e., the relevant node in the whole AST tree of the parsed program), the operator (i.e., UPD, INS, DEL, and MOV) which describes the action performed, and the raw tokens involved in the change.

4.2.3 Tangled Code Changes

Solving a single problem per patch is often considered as a best practice to facilitate maintenance tasks. However, often patches in real-world projects address multiple problems in a patch [97, 222]. Developers often commit bug fixing code changes together with changes unrelated to fix such as functionality enhancements, feature requests, refactorings, or documentation. Such patches are called tangled patches [68] or mixed-purpose fixing commits [171]. Nguyen et al. found that 11% to 39% of all the fixing commits used for mining archives were tangled [171].

Consider the example patch from GWT illustrated in Figure 4.4. The patch is intended to fix the issue[2] that reported a failure in some web browsers when the page is served with a certain mime type (i.e., application/xhtml+xml). The developer fixes the issue by showing a warning when such a mime type is encountered. However, in addition to this change, a typo has been addressed in the commit. Since the typo is not related to the fix, the fixing commit is tangled. There is thus a need to separately consider single code hunks within a commit to allow the pattern inference to focus on finding recurrent atomic changes that are relevant to bug fixing operations.

[2] https://github.com/gwtproject/gwt/issues/676

```
−−− a/dev/core/src/com/google/gwt/dev/shell/GWTShellServlet.java
+++ b/dev/core/src/com/google/gwt/dev/shell/GWTShellServlet.java
@@ −72,6 +72,8 @@

+   private static final String XHTML_MIME_TYPE = "application/xhtml+xml";
  private final Map loadedModulesByName = new HashMap();
  private final Map loadedServletsByClassName = new HashMap();
@@ −110,7 +112,7 @@
      writer.println("<html><body><basefont face='arial'>");
−       writer.println("To launch an an application, specify a URL of the form <code>
     /<i>module</i>/<i>file.html</i></code>");
+       writer.println("To launch an application, specify a URL of the form <code>/<i>
     module</i>/<i>file.html</i></code>");
      writer.println("</body></html>");
    }
@@ −407,6 +409,8 @@
    }
+    maybeIssueXhtmlWarning(logger, mimeType, partialPath);

@@ −755,6 +759,25 @@

+  private void maybeIssueXhtmlWarning(TreeLogger logger, String mimeType,
+      String path) {
+    if (!XHTML_MIME_TYPE.equals(mimeType)) {
+      return;
+    }
+
+    String msg = "File was returned with content−type of \"" + mimeType
+        + "\". GWT requires browser features that are not available to "
+        + "documents with this content−type.";
+
+    int ix = path.lastIndexOf('.');
+    if (ix >= 0 && ix < path.length()) {
+      String base = path.substring(0, ix);
+      msg += " Consider renaming \"" + path + "\" to \"" + base + ".html\".";
+    }
+
+    logger.log(TreeLogger.WARN, msg, null);
+  }
```

Figure 4.4: Tangled commit.

4.3 Approach

FixMiner aims to discover relevant fix patterns from the atomic changes within bug fixing patches in software repositories. To that end, we mine code changes that are similar in terms of context, operations, and the programming tokens that are involved. Figure 5.6 illustrates an overview of the FixMiner approach.

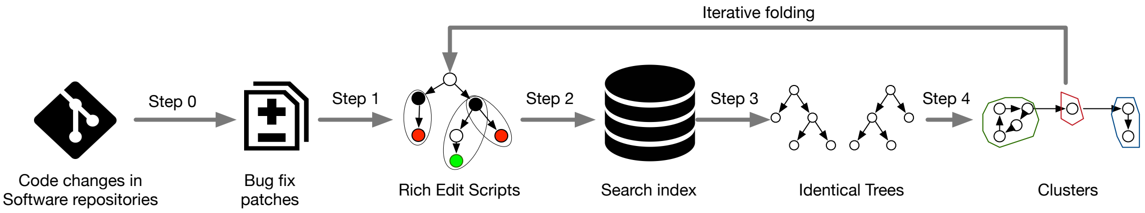


Figure 4.5: The FixMiner Approach. At each iteration, the search index is refined, and the computation of tree similarity is specialised in specific AST information details.

4.3.1 Overview

In Step 0, as an initial step, we collect the relevant bug-fixing patches (cf. Definition 1) from project change tracking systems. Then, in Step 1, we compute a `Rich Edit Script` representation (cf. Section 4.3.3) to describe a code change in terms of the context, operations performed and tokens involved. Accordingly, we consider three specialised tree representations of the `Rich Edit Script` (cf. Definition 2) carrying information about either the impacted AST node types, or the repair actions performed, or the program tokens affected. `FixMiner` works in an iterative manner considering a single specialised tree representation in each pattern mining iteration, to discover similar changes: first, changes affecting the same code context (i.e., on identical abstract syntax trees) are identified; then among those identified changes, changes using the same actions (i.e., identical sequence of operations) are regrouped; and finally, within each group, changes affecting the same tokens set are mined. Therefore, in `FixMiner`, we perform a three-fold strategy, carrying out the following steps in a pattern mining iteration:

- Step 2: We build a search index (cf. Definition 3) to identify the `Rich Edit Script`s that must be compared.
- Step 3: We detect identical trees (cf. Definition 4) by computing the distance between two representations of `Rich Edit Script`s.
- Step 4: We regroup identical trees into clusters (cf. Definition 5).

The initial pattern mining iteration uses `Rich Edit Script`s computed in Step 1 as its input, where the following rounds use clusters of identical trees yielded in Step 4 as their input.

In the following sections, we present the details of Steps 1-4, considering that a dataset of bug-fix patches is available.

4.3.2 Step 0 - Patch Collection

> ***Definition 1*** *(***Patch***) A program patch is a transformation of a program into another program, usually to fix a defect. Let $\mathbb{P}$ be a set of programs, a patch is represented by a pair (p, p'), where $p, p' \in \mathbb{P}$ are programs before and after applying the patch, respectively. Concretely, a patch implements changes in code block(s) within source code file(s).*

To identify bug-fix patches in software repositories projects, we build on the bug linking strategies implemented in the Jira issue tracking software. We use a similar approach to the ones proposed by Fischer et al. [53] and Thomas et al. [223] in order to link commits to relevant bug reports. Concretely, we crawl the bug reports for a given project and assess the links with a two-step search strategy: (i) we check project commit logs to identify bug report IDs and associate the corresponding bug reports to commits; then (ii) we check for bug reports that are indeed considered as such (i.e., tagged as “BUG”) and are further marked as resolved (i.e., with tags “RESOLVED” or “FIXED”), and completed (i.e., with status “CLOSED”).

We further curate the patch set by considering bug reports that are fixed by a single commit. This provides more guarantees that the selected commits are indeed fixing the bugs in a single shot (i.e., the bug does not require supplementary patches [183]). Eventually, we consider only changes that are made on the source code files: changes on configuration, documentation, or test files are excluded.

4.3.3 Step 1 – `Rich Edit Script` Computation

Definition 2 *(`Rich Edit Script`) A `Rich Edit Script` $r \in RE$ represents a patch as a specialised tree of changes. This tree describes which operations are made on a given AST, associated with the code block before patch application, to transform it into another AST, associated with the code block after patch application: i.e., $r : \mathbb{P} \rightarrow \mathbb{P}$. Each node in the tree is an AST node affected by the patch. Every node in `Rich Edit Script` has three different types of information:* **Shape**, **Action**, *and* **Token**.

A bug-fix patch collected in open source change tracking systems is represented in the GNU diff format based on addition and removal of source code lines, as shown in Figure 4.6. This representation is not suitable for fine-grained analysis of changes.

```
// modules. We need to move this code up to a common module.
−        int indexOfDot = namespace.indexOf('.');
+        int indexOfDot = namespace.lastIndexOf('.');
        if (indexOfDot == −1) {
```

Figure 4.6: Patch of fixing bug Closure-93 in Defects4J dataset.

To accurately reflect the change that has been performed, several algorithms have been proposed based on tree structures (such as the AST) [18,33,45,52,56,66,185]. ChangeDistiller [56] and GumTree [52] are state-of-the-art examples of such algorithms which produce edit scripts that detail the operations to be performed on the nodes of a given AST in order to yield another AST corresponding to the new version of the code. In particular, in this work, we selected the GumTree AST differencing tool which has seen a momentum recently in the literature for computing edit scripts. GumTree is claimed to build in a fast, scalable and accurate way the sequence of AST edit actions (a.k.a edit scripts) between the two associated AST representations (the buggy and fixed versions) of a given patch.

```
UPD SimpleName "indexOf" to "lastIndexOf"
```

Figure 4.7: GumTree edit script corresponding to Closure-93 bug-fix patch represented in Figure 4.6.

Consider the example edit script computed by GumTree for the patch of Closure-93 bug from Defects4J illustrated in Figure 4.7. The intended behaviour of the patch is to fix the wrong variable declaration of *indexOfDot* due to a wrong method reference (*lastIndexOf* instead of *indexOf*) of java.lang.String object. GumTree edit script summarises the change as an update operation on an AST node simple name (i.e., an identifier other than a keyword) that is modifying the identifier label (from *indexOf* to *lastIndexOf*).

Although GumTree edit script is accurate in describing the bug-fix operation at a fine-grained level, much of the contextual information describing the intended behaviour of the patch is missing. The information regarding method invocation, the method name (*java.lang.String*), the variable declaration fragment which assigns the value of the method invocation to *indexOfDot*, as well as the type information (*int* for *indexOfDot* - cf. Figure 4.6) that is implied in the variable declaration statement, are all missing in the GumTree edit script. Since such contextual information is lost, the yielded edit script fails to convey the full syntactic and semantic meaning of the code change.

To address this limitation, we propose to enrich GumTree-yielded edit scripts by retaining more contextual information. To that end, we construct a specialised tree structure of the edit scripts which captures the AST-level context of the code change. We refer to this specialised tree structure as `Rich Edit Script`. A `Rich Edit Script` is computed as follows:

Given a patch, we start by computing the set of edit actions (edit script) using GumTree, where the set contains an edit action for each contiguous group of code lines (hunks) that are changed by a patch. In order to capture the context of the change, we reorganise edit actions under new AST (minimal) subtrees building an AST hierarchy. For each edit action in an edit script, we extract a minimal subtree from the original AST tree which has the GumTree edit action as its leaf node, and one of the following predefined node types as its root node: TypeDeclaration, FieldDeclaration, MethodDeclaration, SwitchCase, CatchClause, ConstructorInvocation, SuperConstructorInvocation or any Statement node. The objective is to limit the scope of context to the encompassing statement, instead of going backwards until the compilation unit (cf. Figure 4.2). We limit the scope of parent traversal mainly for two reasons: first, the pattern mining must focus on the program context that is relevant to the change; second, program repair approaches, which `FixMiner` is built for, generally target statement-level fault localisation and patch generation.

Consider the AST differencing tree presented in Figure 4.8. From this diff-tree, GumTree yields the leaf nodes (grey) of edit actions as the final edit script. To build the `Rich Edit Script`, we follow these steps:

i) For each GumTree-produced edit action, we remap it to the relevant node in the program AST;
ii) Then, starting from the *GumTree edit action* nodes, we traverse the AST tree of the parsed program from bottom to top until we reach a node of the *predefined root node* type.
iii) For every *predefined root node* that is reached, we extract the AST subtree between the discovered *predefined root node* down to the leaf nodes mapped to the *GumTree edit actions*.
iv) Finally, we create an ordered[3] sequence of these extracted AST subtrees and store it as `Rich Edit Script`.

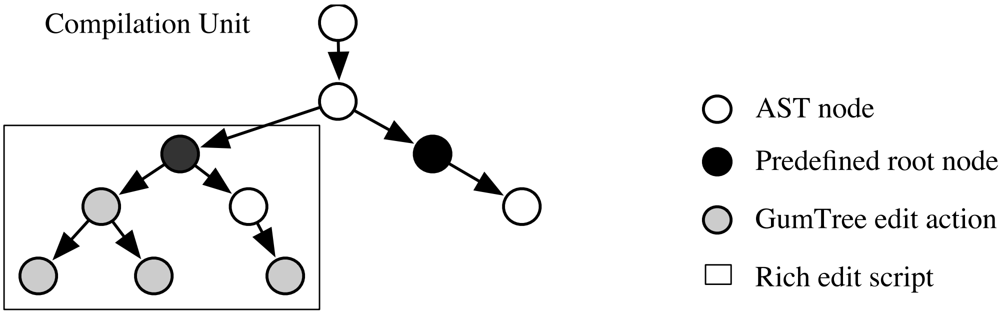


Figure 4.8: Illustration of subtree extraction.

Concretely, with respect to our running example, consider the case of Closure-93 illustrated in Figure 4.6. The construction of the `Rich Edit Script` starts by generating the GumTree edit script (cf. Figure 4.7) of the patch. The patch consists of a single hunk, thus we expect to extract a single AST subtree, which is illustrated in Figure 4.9. To extract this AST subtree, first, we identify the node of the edit action “SimpleName” at position 4 in the AST Tree of program. Then, starting from this node, we traverse backwards the AST tree until we reach the node “VariableDeclarationStatement” at position 1. We extract the AST subtree, by creating a new tree, setting “VariableDeclarationStatement” as the root node of the new tree, and adding the intermediate nodes at positions 2,3 until we reach the corresponding node of the edit action “`UPD SimpleName`” at position 4. We create a sequence, and add the extracted AST subtree to the sequence.

`Rich Edit Scripts` are tree data structures. They are used to represent changes. In order to provide tractable and reusable patterns as input to other APR systems, we define the following string notation (cf. Grammar 4.1) based on syntactic rules governing the formation of correct `Rich Edit Script`.

Figure 4.10 illustrates the computed `Rich Edit Script`. The first line indicates the root node (no dashed line). ‘UPD ’ indicates the action type of the node, VariableDeclarationStatement corresponds to AST node type of the node, tokens between ‘@@’ and ‘@TO@’ contain the corresponding code tokens before the change, whereas tokens between ‘@TO@’ and ‘@AT’ corresponding to new code

[3] The order of AST subtrees follows the order of hunks of the GNU diff format.

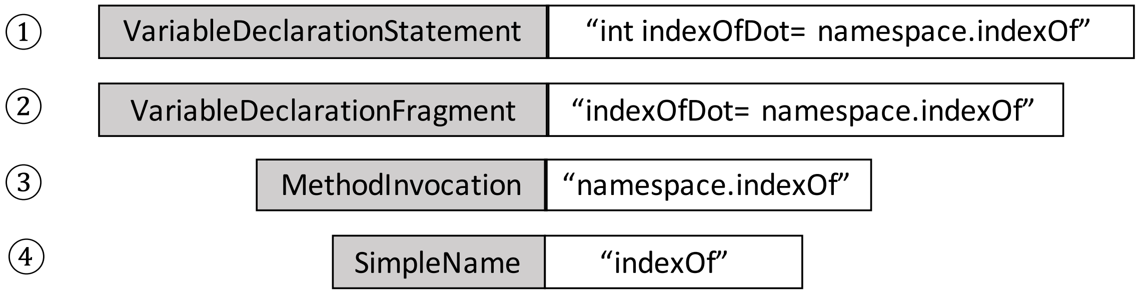

①
VariableDeclarationStatement
"int indexOfDot= namespace.indexOf"
②
VariableDeclarationFragment
"indexOfDot= namespace.indexOf"
③
MethodInvocation
"namespace.indexOf"
④
SimpleName
"indexOf"

Figure 4.9: Excerpt AST of buggy code (Closure-93).

⟨*richEditScript*⟩ ::= ⟨*node*⟩+

⟨*node*⟩ ::= '`---`'* ⟨*move*⟩
| '`---`'* ⟨*delete*⟩
| '`---`'* ⟨*insert*⟩
| '`---`'* ⟨*update*⟩

⟨*move*⟩ ::= '`MOV `' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@TO@`' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@AT@`'

⟨*delete*⟩ ::= '`DEL `' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@AT@`'

⟨*insert*⟩ ::= '`INS `' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@TO@`' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@AT@`'

⟨*update*⟩ ::= '`UPD `' ⟨*astNodeType*⟩ '`@@`' ⟨*tokens*⟩ '`@TO@`' ⟨*tokens*⟩ '`@AT@`'

Grammar 4.1: Notation of `Rich Edit Script`

tokens with the change. The three dashed (`---`) lines indicate a child node. Immediate children nodes contain three dashes while their children add another three dashes (`------`) preserving the parent-child relation.

```
UPD VariableDeclarationStatement@@int indexOfDot = namespace.indexOf('.'); @TO@ int ↩
    indexOfDot = namespace.lastIndexOf('.'); @AT@
---UPD VariableDeclarationFragment@@indexOfDot = namespace.indexOf('.') @TO@ ↩
    indexOfDot = namespace.lastIndexOf('.') @AT@
------UPD MethodInvocation@@namespace.indexOf('.') @TO@ namespace.lastIndexOf('.') @AT@
---------UPD SimpleName@@MethodName:indexOf:['.'] @TO@ ↩
    MethodName:lastIndexOf:['.'] @AT@
```

Figure 4.10: `Rich Edit Script` for Closure-93 patch in Defects4J. ↩ represents the carriage return character which is necessary for presentation reasons.

An edit action node carries the following three types of information: the AST node type (Shape), the repair action (Action), the raw tokens (Token) in the patch. For each of these three information types, we create separate tree representations from the `Rich Edit Script`, named as ShapeTree, ActionTree and TokenTree, each carrying respectively the type of information indicated by its name. Figures 4.11, 4.12, and 4.13 show ShapeTree, ActionTree, and TokenTree, respectively, generated for Closure-93.

```
VariableDeclarationStatement
———VariableDeclarationFragment
——————MethodInvocation
——————————SimpleName
```

Figure 4.11: ShapeTree of Closure-93.

```
UPD root
———UPD child1
——————UPD child1_1
——————————UPD child1_1_1
```

Figure 4.12: ActionTree of Closure-93.

```
@@int indexOfDot = namespace.indexOf('.'); @TO@ int indexOfDot = namespace.la...
———@@indexOfDot = namespace.indexOf('.') @TO@ indexOfDot = namespace.lastInde...
——————@@namespace.indexOf('.') @TO@ namespace.lastIndexOf('.')
——————————@@MethodName:indexOf:['.'] @TO@ MethodName:lastIndexOf:['.']
```

Figure 4.13: TokenTree of Closure-93.

4.3.4 Step 2 – Search Index Construction

> ***Definition 3*** *(Search Index) To reduce the effort of matching similar patches, a search index (SI) is used to confine the comparison space. Each fold ({Shape, Action, Token}) defines a search index:* SI_{Shape}*,* SI_{Action}*, and* SI_{Token}*, respectively. Each is defined as* $SI_* : Q_* \rightarrow 2^{RE}$*, where* Q *is a query set specific to each fold and* $* \in \{Shape, Action, Token\}$*.*

Given that `Rich Edit Scripts` are computed for each hunk in a patch, they are spread inside and across different patches. A direct pairwise comparison of these `Rich Edit Scripts` would lead to a combinatorial explosion of the comparison space. In order to reduce this comparison space and enable a fast identification of `Rich Edit Scripts` to compare, we build search indices. A search index is a set of comparison sub-spaces created by grouping the `Rich Edit Scripts` with criteria that depend on the information embedded the used tree representation (Shape, Action, Token) for the different iterations.

The search indices are built as follows:

"Shape" search index. The construction process takes the ShapeTree representations of the `Rich Edit Scripts` produced by Step 1 as input, and groups them based on their tree structure in terms of AST node types. Concretely, `Rich Edit Scripts` having the same root node (e.g., IfStatement, MethodDeclaration, ReturnStatement) and same depth are grouped together. For each group, we create a comparison space by enumerating the pairwise combinations of the group members. Eventually, the "Shape" search index is built by storing an identifier per group, denoted as root node/depth (e.g., IfStatement/2, IfStatement/3, MethodDeclaration/4), and a pointer to its comparison space (i.e., the pairwise combinations of its members).

"Action" search index. The construction process follows the same principle as for "Shape" search index, except that the regrouping is based on the clustering output of ShapeTrees. Thus, the input is formed by ActionTree representations of the `Rich Edit Scripts` and the group identifier for each comparison space is generated as node/depth/ShapeTreeClusterId (e.g., IfStatement/2/1, MethodDeclaration/2/2) where ShapeTreeClusterId represents the id of the cluster yielded by the clustering (Steps 3-4) based on the ShapeTree information. Concretely, this means that the "Action" search index is built on groups of trees having the same shape.

"Token" search index. The construction process follows the same principle as for "Action" search index, using this time the clustering output of ActionTrees. Thus, the input is formed by TokenTree representations of the `Rich Edit Scripts` and the group identifier for each comparison space is generated as node/depth/ShapeTreeClusterId/ActionTreeClusterId (e.g., IfStatement/2/1/3, MethodDeclaration/2/2/1) where ActionTreeClusterId represents the id of the cluster yielded by the clustering (Steps 3-4) based on the ActionTree information.

4.3.5 Step 3 – Tree Comparison

> ***Definition 4*** *(Pair of identical trees) Let* $a = (r_i, r_j) \in R_{identical}$ *be a pair of* `Rich Edit Script` *specialised tree representations if* $d(r_i, r_j) = 0$*, where* $r_i, r_j \in RE$ *and* d *is a distance function.* $R_{identical}$ *is a subset of* $RE \times RE$*.*

The goal of tree comparison is to find identical tree representations of `Rich Edit Scripts` for a given fold. There are several straightforward approaches for checking whether two `Rich Edit Scripts` are identical. For example, syntactical equality could be used. However, we aim at making `FixMiner` a flexible and extensible framework where future research may tune threshold values for defining similar trees. Thus, we propose a generic approach for comparing `Rich Edit Scripts`, taking into account the diversity of information to compare for each specialised tree representation. To that end, we compute tree edit distances for the three representations of `Rich Edit Scripts` separately. The tree edit distance is defined as the sequence of edit actions that transform one tree into another. When the edit distance is zero (i.e., no operation is necessary to transform one tree to another), the trees are considered as identical. In Algorithm 1, we define the steps to compare `Rich Edit Scripts`.

The algorithm starts by retrieving the identifiers from the search index SI corresponding to the $fold$. An identifier is a pointer to a comparison sub-space that contains pairwise combinations of tree representation of `Rich Edit Scripts` to compare (cf. Section 4.3.4). Concretely, we restore the `Rich Edit Scripts` of a given pair from the cache, and their corresponding specialised tree representation according to the $fold$: At the first iteration, we consider only trees denoted ShapeTrees, whereas in the second iteration we focus on ActionTrees and TokenTrees for the third iteration. We compute the edit distance between the restored trees in two distinct ways.

- In the first two iterations (i.e., Shape and Action), we leverage the edit script algorithm of GumTree [51, Section 3] again. We compute the edit distance by simply invoking GumTree on restored trees as input, given that `Rich Edit Scripts` are indeed AST subtrees that are compatible with GumTree. Concretely, GumTree takes the two AST trees as input and generates a sequence of edit actions (a.k.a edit script) that transform one tree into another, where the size of the edit script represents the edit distance between the two trees.
- For the third iteration (i.e., Token), since the relevant information in the tree is text, we use a text distance algorithm (Jaro-Winkler [75, 246]) to compute the edit distance between two tokens extracted from the trees. We use the implementation of Jaro-Winkler edit distance from Apache Commons Text library[4], which computes the Jaro-Winkler edit distance of two strings d_w as defined in Equation 4.1. The equation consists of two components; Jaro's original algorithm (j_{sim}) and Winkler's extension(w_{sim}). The Jaro similarity is the weighted sum of the percentage of matched characters c from each file and transposed characters t. Winkler increased this measure for matching initial characters, by using a prefix scale p that is set to 0.1 by default, which gives more favourable ratings to strings that match from the beginning for a set prefix length l. The algorithm produces a similarity score (w_{sim}) between 0.0 to 1.0, where 0.0 is the least likely and

[4] `https://commons.apache.org/proper/commons-text/`

Algorithm 1: `Rich Edit Script` Comparison.

```
input  : $SI$: Search Index where $SI \in \{SI_{Shape}, SI_{Action}, SI_{Token}\}$
input  : $fold \in \{Shape, Action, Token\}$
input  : $threshold$: Set to 0 to retrieve identical trees.
output : $R_{\text{identical}}$: A set of pairs tagged to be identical

Function main (SI,fold)
    $R_{\text{identical}} \leftarrow \emptyset$
    $I \leftarrow SI$.getIdentifiers()              /* $I$: list of identifiers in the index */
    foreach $i \in I$ do
        $R \leftarrow SI$.getPairs($i$)  /* $R$: list of tree pairs to compare for identifier $i$ */
        foreach $a \in R$ do
            if compareTree(a,fold) then
                $R_{\text{identical}}$.add($a$)       /* add if $a$ is a pair of identical trees */
    return $R_{\text{identical}}$
Function compareTree(a,fold)
    ($sTree1$,$sTree2$) $\leftarrow$ specializedTree($a$,$fold$)
    if $Fold$ != $Token$ then
        $editActions \leftarrow$ GumTree($sTree1$, $sTree2$)
        $editDistance \leftarrow$ size($editActions$)
    else
        $tokens1$,$tokens2 \leftarrow$ getTokens($sTree1$,$sTree2$)
        $editDistance \leftarrow d_w$($tokens1$,$tokens2$)        /* $d_w$: Jaro-Winkler distance */
    if $editDistance <= threshold$ then
        return true
    else
        return false
Function specializedTree(a,fold)
    ($eTree1$, $eTree2$) $\leftarrow$ getRichEditScripts($a$)   /* restore Rich Edit Scripts of a given
     pair from the cache */
    if $fold == Shape$ then
        $sTree1$,$sTree2 \leftarrow$ getASTNodeTrees($eTree1$, $eTree2$)
    else if $fold == Action$ then
        $sTree1$,$sTree2 \leftarrow$ getActionTrees($eTree1$, $eTree2$)
    else
        $sTree1$,$sTree2 \leftarrow$ getTokenTrees($eTree1$, $eTree2$)        /* $fold$ == $Token$ */
    return ($sTree1$,$sTree2$)
```

1.0 is a positive match. Finally, this similarity score is transformed to distance (d_w).

$$
\begin{aligned}
&d_w(s_1, s_2) = 1 - w_{sim}(s_1, s_2)\\
&w_{sim}(s_1, s_2) = j_{sim}(s_1, s_2) + l * p(1 - j_{sim}(s_1, s_2))\\
&j_{sim}(s_1, s_2) = \begin{cases} 0 & \text{if } c = 0;\\ \frac{1}{3}\left(\frac{c}{\operatorname{abs} s_1} + \frac{c}{\operatorname{abs} s_2} + \frac{c-t}{c}\right) & otherwise. \end{cases}
\end{aligned}
\tag{4.1}
$$

l: The number of agreed characters at the beginning of two strings.
p: is a constant scaling factor for how much the score is adjusted upwards for having common prefixes, which is set to 0.1 in Winkler's work [246].

As the last step of comparison, we check the edit distance of the tree pair and tag the pairs having the distance zero as identical pairs, since the distance zero implies that no operation is necessary to transform one tree to another, or for the third fold ($Token$) the tokens in the tree are the same. Eventually, we store and save the set of identical tree pairs produced in each iteration, which would be used in Step 4.

4.3.6 Step 4 – Pattern Inference

> ***Definition 5*** *(Pattern) Let g be a graph in which nodes are elements of RE and edges are defined by $R_{identical}$.*
> *g consists of a set of connected subgraphs SG (i.e., clusters of specialised tree representations of `Rich Edit Scripts`) where sg_i and sg_j are disjoint $\forall sg_i, sg_j \in SG$. A pattern is defined by $sg_i \in SG$ if sg_i has at least two nodes (i.e., there are recurrent trees).*

Finally, to infer patterns, we resort to clustering of the specialised tree representations of `Rich Edit Scripts`. First, we start by retrieving the set of identical tree pairs produced in Step 3 for each iteration. Following Algorithm 2, we extract the corresponding specialised tree representations according to the fold (i.e., ShapeTrees, ActionTrees, TokenTrees) since the trees are identical only in a given fold. In order to find groups of trees that are identical among themselves (i.e., clusters), we leverage graphs. Concretely, we implement a clustering process based on the theory of connected components (i.e., subgraph) identification in a graph [209]. We create an undirected graph from the list of tree pairs, where the nodes of the graph are the trees and the edges represent trees that are associated (i.e., identical tree pairs). From this graph, we identify clusters as the subgraphs, where each subgraph contains a group of trees that are identical among themselves and disjoint from others.

Algorithm 2: Clustering based on subgraph identification.

```
input  : $R_{\text{identical}}$: A list of identical Rich Edit Script pairs
input  : $fold \in \{Shape, Action, Token\}$
output : $C$: A list of clusters
Function main($R_{identical}$, $fold$)
    $C \leftarrow \emptyset$
    $TP \leftarrow$ getTreePairs($R_{\text{identical}}$, $fold$)
    $E \leftarrow$ transformPairsToEdges($TP$)      /* E: edges created from tree pairs $TP$ */
    $g \leftarrow$ createGraph($E$)
    $SG \leftarrow$ g.connectedComponents()    /* $SG$: list of subgraphs found in graph $g$ */
    foreach $sg$ in $SG$ do
        $c \leftarrow$ s.nodes()          /* c: cluster formed from the nodes of subgraph $sg$ */
        $C$.add($c$)
    return $C$
Function getTreePairs($R_{identical}$, $fold$)
    $P \leftarrow \emptyset$                                      /* P: list of tree pairs */
    foreach $a$ in $R_{identical}$ do
        ($eTree1$, $eTree2$) $\leftarrow$ getRichEditScripts($a$)      /* restore Rich Edit Scripts of a
         given pair from the cache */
        if $fold == Shape$ then
            $sTree1$,$sTree2$ $\leftarrow$ getASTNodeTrees($eTree1$, $eTree2$)
        else if $fold == Action$ then
            $sTree1$,$sTree2$ $\leftarrow$ getActionTrees($eTree1$, $eTree2$)
        else
            $sTree1$,$sTree2$ $\leftarrow$ getTokenTrees($eTree1$, $eTree2$)          /* $fold$ == $Token$ */
        $P$.add($sTree1$,$sTree2$)
    return $P$
```

A cluster contains a list of `Rich Edit Scripts` sharing a common specialised tree representation according to the $fold$. Finally, a cluster is qualified as a pattern when it has at least two members. The patterns for each $fold$ are defined as follows:

Shape patterns. The first iteration attempts to find patterns in the ShapeTrees associated with developer patches. We refer to them as Shape patterns, since they represent the shape of the changed code in a structure of the tree in terms of node types. Thus, they are not fix patterns per se, but rather the context in which the changes are recurrent.

Action patterns. The second iteration considers samples associated to each shape pattern and attempts to identify reoccurring repair actions from their ActionTrees. This step produces patterns that are relevant to program repair as they refer to recurrent code change actions. Such patterns can indeed be matched to dissection studies performed in the literature [211]. We will refer to Action patterns as the sought fix patterns. Nevertheless, it is noteworthy that, in contrast with literature fix patterns, which can be generically applied to any matching code context, our Action patterns are specifically mapped to a code shape (i.e., a shape pattern) and is thus applicable to specific code contexts. This constrains the mutations to relevant code contexts, thus yielding more likely precise fix operations.

Token patterns. The third iteration finally considers samples associated to each action pattern and attempts to identify more specific patterns with respect to the tokens available. Such token-specific patterns, which include specific tokens, are not suitable for implementation into pattern-based automated program repair systems from the literature. We discuss, however, their use in the context of deriving collateral evolutions (cf. Section 4.5.2.1).

4.4 Experimental Evaluation

We now provide details on the experiments that we carry out for `FixMiner`. Notably, we discuss the dataset, and present the implementation details. Then, we overview the statistics on the mining steps, and eventually enumerate the research questions for the assessment of `FixMiner`.

4.4.1 Dataset

We collect code changes from 44 large and popular open-source projects from Apache-Commons, JBoss, Spring and Wildfly communities with the following selection criteria: we focused on projects (1) written in Java, (2) with publicly available bug reports, (3) having at least 20 source code files in at least one of its versions; finally, to reduce selection bias, (4) we choose projects from a wide range of categories - middleware, databases, data warehouses, utilities, infrastructure. This is a process similar to Bench4bl [120]. Table 6.1 details the number of bug fixing patches that we considered in each project. Eventually, our dataset includes 11 416 patches.

4.4.2 Implementation Choices

We recall that we have made the following parameter choices in the `FixMiner` workflow:

- The “Shape” search index considers only `Rich Edit Scripts` having a depth greater than 1 (i.e., the AST sub-tree should include at least one parent and one child).
- Comparison of `Rich Edit Scripts` is designed to retrieve identical trees (i.e., tree edit distance is 0).

Table 4.2: Dataset.

Community	Project	# Patches	Project	# Patches
Apache	camel	1366	commons codec	11
	commons collections	56	commons compress	73
	commons configuration	89	commons crypto	9
	commons csv	18	common io	58
	hbase	2169	hive	2641
JBoss	entesb	15	jbmeta	14
Spring	amqp	89	android	5
	batch	224	batchadm	11
	datacmns	151	datagraph	19
	datajpa	112	datamongo	190
	dataredis	65	datarest	91
	ldap	26	mobile	11
	roo	414	sec	304
	secoauth	66	sgf	35
	shdp	35	shl	11
	social	14	socialfb	12
	socialli	2	socialtw	9
	spr	1098	swf	84
	sws	101		
Wildfly	ely	217	swarm	131
	wfarq	8	wfcore	547
	wfly	802	wfmp	13
Total				**11416**

4.4.3 Statistics

`FixMiner` is a pattern mining approach to produce fix patterns for program repair systems. Its evaluation (cf. Section 4.5) will focus on evaluating the relevance of the yielded patterns. Nevertheless, we provide statistics on the mining process to provide a basis of discussion on the implications of `FixMiner`'s design choices.

4.4.3.1 Search Indices

`FixMiner` mines fix patterns through comparison of hunks (i.e., contiguous groups of code lines). 11 416 patches in our database are eventually associated with 41 823 hunks. A direct pairwise comparison of these hunks would lead to 874 560 753 tree comparison computations. The combinatorial explosion of the comparison space is overcome by building search indices as previously described in Section 4.3.4. Table 4.3 shows the details on the search indices built for each fold in the `FixMiner` iterations. From the 874+ million tree pairs to be compared (i.e., C^2_{41823}), the construction of the Shape index (implements criteria on the tree structure to focus on comparable trees) lead to 670 relevant comparison sub-spaces yielding a total of only 12+ million tree comparison pairs. This represents a reduction of 98% of the comparison space. Similarly, the Action index and the Token index reduce the associated comparison spaces by 88% and 72% respectively.

4.4.3.2 Clusters

We infer patterns by considering recurrence of trees: the clustering process groups together only tree pairs that are identical among themselves. Table 4.4 overviews the statistics of clusters yielded for

Table 4.3: Comparison space reduction.

Search Index	# of hunks (in fold)	# Comparison sub-spaces	# Tree comparison pairs
Shape	41 823	670	12 601 712
Action	25 290	2 457	1 427 504
Token	6759	411	401 980

the different iterations: Shape patterns (which represent code contexts) are the most diverse. Action patterns (which represent fix patterns that are suitable as inputs for program repair systems) are substantially less numerous. Finally, Token patterns (which may be codebase-specific) are significantly fewer. We recall that we consider all possible clusters as long as it includes at least 2 elements. A practitioner may however decide to select only large clusters (i.e., based on a threshold).

Table 4.4: Statistics on clusters.

Pattern	# Trees (clustering input)	# Corresponding change hunks	# Clusters
Shape	1 370 406	25 290	2947
Action	628 531	6 759	428
Token	18 471	1 562	326

Because `FixMiner` considers code hunks as the unit for building `Rich Edit Scripts`, a given pattern may represent a repeating context (i.e., Shape pattern) or change (i.e., Action or Token pattern) that is only *part* of the patch (i.e., this patch includes other change patterns) or that is the *full* patch (i.e., the whole patch is made of this change pattern). Table 4.5 provides statistics on partial and full patterns. The numbers represent the disjoint sets of patterns that can be identified as always full or as always partial. Patterns that may be *full* for a given patch but *partial* for another patch are not considered. Overall, the statistics indicate that, from our dataset of over 40 thousand code hunks, only a few (e.g., respectively 278 and 7 120 hunks) are associated with patterns that are always *full* or always *partial* respectively. In the remaining cases, the pattern is associated with a code hunk that may form alone the patch or tangled with other code. This suggests that `FixMiner` is able to cope with tangled changes during pattern mining.

Table 4.5: Statistics on Full vs Partial patterns.

	Partial patterns			Full patterns		
	# Patterns	# Patch	# Hunk	# Patterns	# Patch	# Hunk
Shape	1358	3140	7120	120	223	278
Action	124	554	1153	10	20	25
Token	75	148	277	14	22	32

Similarly, we investigate how the patterns are spread among patches. Indeed, a pattern may be found because a given patch has made the same change in several code hunks. We refer to such pattern as *vertical*. In contrast, a pattern may be found because the same code change is spread across several patches. We refer to such pattern as *horizontal*. Table 4.6 shows that vertical and horizontal patterns occur in similar proportions for Shape and Action patterns. However, Token patterns are significantly more vertical than horizontal (65 vs 224). This is in line with studies of collateral evolutions in Linux, which highlight large patches making repetitive changes in several locations at once [177] (i.e., collateral evolutions are applied through vertical patches).

Table 4.6: Statistics on Pattern Spread.

	Vertical			Horizontal		
	# Patterns	# Patch	# Hunk	# Patterns	# Patch	# Hunk
Shape	881	881	2432	1194	3808	3808
Action	148	148	488	132	574	574
Token	224	224	709	65	170	170

* A pattern can simultaneously be vertical (when it is associated to several changes in hunks of the same patch) and horizontal (when it appears as well within other patches).

4.4.4 Research Questions

The assessment experiments are performed with the objective of investigating the usefulness of the patterns mined by `FixMiner`. To that end, we focus on the following research questions (RQs):

RQ-1 Is automated patch clustering of `FixMiner` consistent with human manual dissection?
RQ-2 Are patterns inferred by `FixMiner` compatible with known fix patterns?
RQ-3 Are the mined patterns effective for automated program repair?

4.5 Results

4.5.1 RQ1: Comparison of `FixMiner` Clustering against Manual Dissection

Objective. We propose to assess the relevance of the clusters yielded by `FixMiner` in terms of whether they represent patterns which practitioners would view as recurrent changes that are indeed relevant to the patch behaviour. In the previous section, the statistics showed that several changes are recurrent and are mapped to `FixMiner`'s clusters. In this RQ, we validate whether they are relevant to the practitioner's viewpoint. For example, if `FixMiner` were not leveraging AST information, removal of blank lines would have been seen as a recurrent change (hence a pattern); however, a practitioner would not consider it as relevant.

Protocol. We consider an oracle dataset of patches with change patterns that are labelled by humans. Then we associate each of these patches to the relevant clusters mined by `FixMiner` on our combined study datasets. This way, we ensure that the clustering does not overfit to the oracle dataset labelled by humans. Eventually, we check whether each set of patches (from the oracle dataset) that are associated with a given `FixMiner` cluster, consists of patches having the same labels (from the oracle).

Oracle. For our experiments, we leverage the manual dissection of Defects4J [78] provided by Sobreira et al. [211].

This oracle dataset associates the developer patches of 395 bugs in the Defects4J datasets with 26 repair pattern labels (one of which is being "Not classified").

Results. Table 4.7 provides statistics that describe the proportion[5] of `FixMiner`'s patterns that can be associated to change patterns in the Defects4J patches.

[5] In this experiment, we excluded 34 patches from Defects4J dataset which affect more than 1 file.

Table 4.7: Proportion of shared patterns between our study dataset and Defects4J.

	Study dataset		Defects4J	
	# corresponding hunks	# Patterns	# corresponding hunks	# Patterns
Shape	25272	2947	479	214
Action	6755	428	103	37
Token	1562	326	23	13

4.5.1.1 Diversity

We check the number of patterns that can be found in our study dataset and Defects4J. In absolute numbers, Defects4J patches include a limited set of change patterns (i.e., $\sim 7\% = \frac{214}{2947}$) in comparison to what can be found in our study dataset.

4.5.1.2 Consistency

We check for consistency of `FixMiner`'s pattern mining by assessing whether all Defects4J patches associated with a `FixMiner` cluster are indeed sharing a common dissection pattern label. We have found that the clustering to be consistent for $\sim 78\% = \frac{166}{214}$, $\sim 73\% = \frac{27}{37}$ and $\sim 92\% = \frac{12}{13}$ of Shape, Action and Token clusters respectively.

> **RQ1-Consistency ▶** *`FixMiner` can produce patterns that are matching patches that are labelled similarly by humans. The patterns are thus largely consistent with manual dissection.*

4.5.1.3 Granularity

The human dissection provides repair pattern labels for a given patch. Nonetheless, the label is not specifically associated with any of the various changes in the patch. `FixMiner`, however, yields patterns for code hunks. Thus, while `FixMiner` links a given hunk to a single pattern, the dissection data associates several patterns to a given patch. We investigate the granularity level with respect to human-provided patterns. Concretely, several patterns of `FixMiner` can actually be associated (based on the corresponding Defects4J patches) to a single human dissection pattern. Consider the example cases in Table 4.8. Both patches consist of nested InfixExpression under the IfStatement. The first `FixMiner` pattern indicates that the change operation (i.e., update operator) should be performed on the children InfixExpression. On the other hand, the second pattern implies a change operation in the parent InfixExpression. Thus, eventually, `FixMiner` patterns are finer-grained and associate the example patches to two distinct patterns, each pointing the precise node to update, while manual dissection considers them under the same coarse-grained repair pattern.

We have investigated the differences between `FixMiner` patterns and dissection labels and found several granularity mismatches similar to the previous example: `condBlockRetAdd` (*condition block addition with return statement*) from manual dissection is associated to 14 fine-grained Shape patterns of `FixMiner`: this suggests that the repair-potential of this pattern could be further refined depending on the code context. Similarly, `expLogicMod` (*logic expression modification*), is associated to 2 separate Action patterns (see Table 4.8) of `FixMiner`: this suggests that the application of this repair pattern can be further specialised to reduce the repair search space and the false positives.

Overall, we found in total 37, 3 and 1 dissection repair patterns are further refined into several `FixMiner`'s Shape, Action and Token patterns respectively.

Table 4.8: Granularity example to FixMiner mined patterns.

	Pattern	Example patch from FixMiner dataset

FixMiner	UPD IfStatement
	———UPD InfixExpression
	——————UPD InfixExpression
	—————————UPD Operator
		@@ −83,7 +83,7 @@ public BoundedInputStrea ...
		@Override
		public int read() throws IOException {
		− if (max >= 0 && pos == max) {
		+ if (max >= 0 && pos >= max) {
		return −1;

Dissection [211]	Logic expression modification Single Line	

FixMiner	UPD IfStatement
	———UPD InfixExpression
	——————UPD Operator
		@@ −145,7 +145,7 @@ private void moveFile(Path s ...
		private Path createTargetPath(Path targetPath ...
		Path deletePath = null;
		Path mkDirPath = targetPath.getParent();
		− if (mkDirPath != null & !fs.exists(mkDirPath)) {
		+ if (mkDirPath != null && !fs.exists(mkDirPath)) {
		Path actualPath = mkDirPath;

Dissection [211]	Logic expression modification Single Line	

RQ1-Granularity ▶ *We observe that manually-dissected patterns are more coarse-grained compared to FixMiner's patterns.*

4.5.1.4 Assessment of FixMiner's patterns with respect to associated bug reports

Beyond assessing the consistency of FixMiner's patterns based on human-built oracle dataset of labels, we further propose to investigate the relevance of the patterns in terms of the semantics that can be associated to the intention of the changes. To that end, we consider bug reports associated with patches as a proxy to characterise the intention of the code changes. We expect bug reports sharing textual similarity to be addressed by patches that are syntactically similar. This hypothesis drives the entire research direction on Information retrieval-based bug localisation [120].

Figure 4.14 provides the distribution of pairwise bug report (textual) similarity values for the bug reports corresponding to patches associated to each cluster. For clear presentation, we focus on the top-20 clusters (in terms of size). We use TF-IDF to represent each bug report as a vector, and leverage Cosine similarity to compute similarity scores among vectors. The represented boxplots display all pairwise bug report similarity values, including outliers. Although for Shape and Action patterns, the similarities are near 0 for all clusters, we note that there are fewer outliers for Action patterns. This suggests a relative increase in the similarity among bug reports. As expected, similarity among bug reports is the highest with Token patterns.

4.5.2 RQ2: Compatibility between FixMiner's patterns and APR literature patterns

Objective. Given that FixMiner aims to automatically produce fix patterns that can be used by automated program systems, we propose to assess whether the yielded patterns are compatible with patterns in the literature.

Protocol. We consider the set of patterns used by literature APR systems and compare them against FixMiner's patterns. Concretely, we systematically try to map FixMiner's patterns with patterns in the literature. To that end, we rely on the comprehensive taxonomy of fix patterns proposed by Liu et al. [133]: if a given FixMiner pattern can be mapped to a type of change in the taxonomy, then this pattern is marked as *compatible* with patterns in the literature.

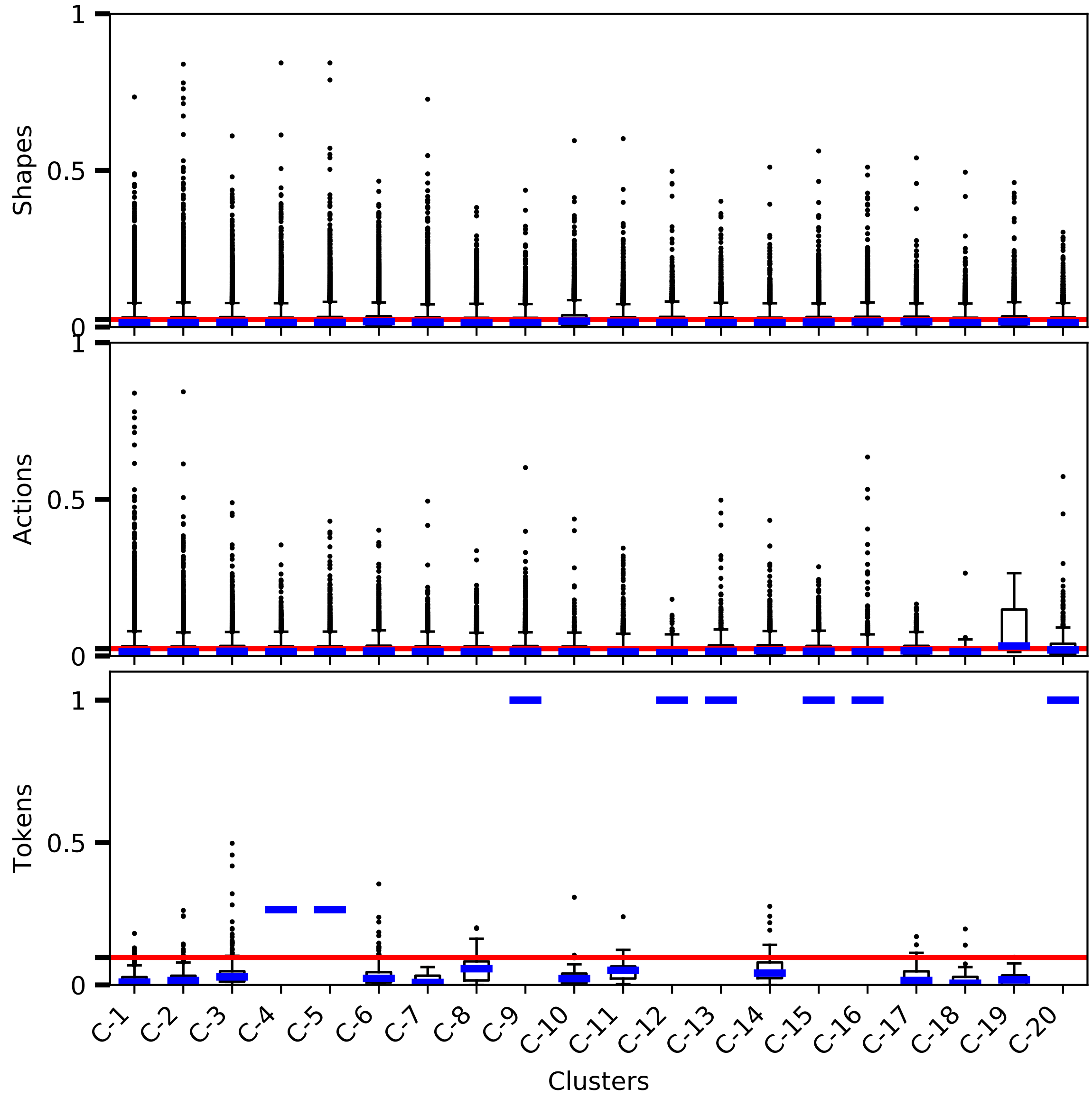


Figure 4.14: Distribution of pairwise bug report similarity. Note: A red line represents an average similarity for all bug reports in fold, and blue line represents average similarity bug reports within a cluster.

Recall that, as described earlier, fix patterns used by APR tools abstract changes at the form of `FixMiner`'s Action patterns (Section 4.3 - Step 4). In the absence of common language for specifying patterns, the comparison is performed manually. For the comparison, we do not conduct exact mapping between literature patterns and the ones yielded by `FixMiner` as fix patterns yielded by `FixMiner` have more context information. We rather consider whether the context information yielded by `FixMiner` patterns matches with the context of literature patterns. We discuss the related threats to validity in Section 4.6.1. Given that the assessment is manual and thus time-consuming, we limit the comparisons to the top 50 patterns (i.e., Action patterns) yielded by `FixMiner`.

Oracle. We build on the patterns enumerated by Liu et al. [133] who systematically reviewed fix patterns used by Java APR systems in the literature. They summarised 35 fix patterns in GNU format, which we refer to for comparing against `FixMiner` patterns.

Results. Overall, among the 35 fix patterns used by the total of 11 studied APR systems, 16 patterns are also included in the fix patterns (i.e., Action patterns) yielded by `FixMiner` when mining our study dataset. We recall that these patterns are often manually inferred and specified by researchers for their APR tools. Table 4.9 illustrates examples of `FixMiner`'s fix patterns associated with some of the patterns used in literature. We note that fix patterns identified by `FixMiner` are specific (e.g., for `FP4: Insert Missed Statement`, the corresponding `FixMiner`'s fix pattern specifies which type of statement must be inserted).

Table 4.9: Example `FixMiner` fix-patterns associated to APR literature patterns.

Patterns enumerated by Liu et al. [133]	Example fix pattern from `FixMiner` (*)
FP2. Insert Null Pointer Checker	INS IfStatement — INS InfixExpression —— INS SimpleName —— INS Operator —— INS NullLiteral — INS ReturnStatement —— INS NullLiteral
FP4. Insert Missed Statement	INS ExpressionStatement —INS MethodInvocation ——INS SimpleName
FP7. Mutate Data Type	UPD CatchClause — UPD SingleVariableDeclaration —— UPD SimpleType
FP9. Mutate Literal Expression	UPD FieldDeclaration — UPD VariableDeclarationFragment —— UPD StringLiteral
FP10. Mutate Method Invocation Expression	UPD ExpressionStatement — UPD MethodInvocation —— UPD SimpleName ——— INS SimpleName
FP11. Mutate Operators	UPD IfStatement — UPD InfixExpression —— UPD Operator
FP12. Mutate Return Statement	UPD ReturnStatement — UPD MethodInvocation —— UPD SimpleName

* **Complete list of 16 Fix Patterns from literature that match `FixMiner`'s patterns:** FP2. Insert Null Pointer Checker (i.e., 2.1, 2.2 and 2.5), FP3. Insert Range Checker, FP4. Insert Missed Statement (i.e., 4.1), FP7. Mutate Data Type (i.e., 7.1), FP9. Mutate Literal Expression (i.e., 9.1), FP10. Mutate Method Invocation Expression (i.e., 10.1, 10.2, 10.3, and 10.4), FP11. Mutate Operators (i.e., 11.1), FP12. Mutate Return Statement, FP13. Mutate Variable (i.e., 13.1), FP14. Move Statement and FP15. Remove Buggy Statement (i.e., 15.1).

Table 4.10 illustrates the proportion of `FixMiner`'s patterns that are compatible with patterns in the literature. In this comparison, we select the top-50 fix patterns yielded by `FixMiner` and verify their presence within the fix patterns used in the APR systems.

Table 4.10: Compatibility of Patterns: `FixMiner` vs Literature Patterns.

PAR	**HDRepair**	**ssFix**	**ELIXIR**	**S3**	**NPEfix**	**SketchFix**	**SOFix**	**Genesis**	**CapGen**	**SimFix**	**AVATAR**
7/16	7/12	6/34	8/11	3/4	1/9	5/6	9/12	1/108	12/30	8/16	6/13

We provide x/y numbers: x is the number of fix patterns in the corresponding APR tool that are mined by `FixMiner`; y is the number of fix patterns used by the corresponding APR tool.

We observed that

- 7 patterns are compatible with fix patterns that are mined manually from bug fix patches (i.e., fix patterns in PAR [88]).
- between 1 and 8 patterns are compatible with researcher-predefined fix patterns used in ssFix [253], ELIXIR [200], S3 [110], NEPfix [48], and SketchFix [72], respectively.
- 7 patterns are compatible with fix pattern mined from history bug fixes by HDRepair [114], 9 patterns are compatible with fix patterns mined from StackOverflow by SOFix [135], and 1 fix pattern is compatible with 1 fix pattern mined by Genesis [137] that focuses on mining fix patterns for three kinds of bugs.
- 12 and 8 patterns are compatible with the patterns used by CapGen [241] and SimFix [76], respectively, which extract patterns in a statistic way similar to the empirical studies of bug fixes [130, 151].
- 6 patterns are compatible with the fix patterns used in AVATAR [132], which are presented in a study for inferring fix patterns from FindBugs [70] static analysis violations [129].

> **RQ2▶** *`FixMiner` effectively yields Action patterns that are compatible for 16 over 35 patterns used in the literature of pattern-based program repair.*

4.5.2.1 Manual (but Systematic) Assessment of Token patterns.

Action and Token patterns are the two types of patterns that relate to code changes. In the assessment scenario above, we only considered Action patterns since they are the most appropriate for comparison with the literature patterns. We now focus on Token patterns to assess whether our hypothesis on their usefulness for deriving collateral evolutions holds (cf. Section 4.3 - Step 4). To that end, we consider the various Token clusters yielded by `FixMiner` and manually verify whether the recurrent change (i.e., the pattern) is relevant (i.e., a human can explain whether the intentions of the changes are the same). Eventually, if the pattern is validated, it should be presented as a generic/semantic patch [9, 177] written in SmPL[6].

In Table 4.11, we list some of the patches that we found to be relevant. Among the top 50 Token patterns investigated, 12 patterns correspond to a modifier change, 4 patterns target changes in logging methods, and 1 pattern is about fixing the infix operator (e.g., `>` → `>=`). The remaining cases mainly focus on changes that complete the implementation of code *finally* block logic (e.g., missing call to *closeAll* for opened files), changes in Exception handling, updates to wrong parameters passed to method invocations, as well as wrong method invocations. As mentioned earlier, these patterns are spread mostly vertically (i.e., change is recurrent in several code hunks of a given patch) and the semantic behaviour of these patterns are specific to project nature.

Overall, our manual investigations on the top 50 Token patterns confirm that many of the recurrent changes associated with specific tokens are indeed relevant. We even found several cases where collateral evolution changes are regrouped to form a pattern as exhibited by the corresponding pattern example presented in Figure 4.15. In this example, we illustrate the pattern using the SmPL specification language, which was designed for specifying collateral evolutions. This finding suggests that `FixMiner` can be leveraged to systematically mined collateral evolutions in the form of Token patterns which could be automatically rewritten as semantic patches in SmPL format. This endeavour is however, out of the scope of this work and will be investigated in future work.

[6] Semantic Patch Language

Table 4.11: Example changes associated to `FixMiner` mined patterns.

Semantic Behaviour of Pattern	Example change in developer patch
Missing field modifier	− private boolean closed = true; + private volatile boolean closed = true;

Wrong Log level	} catch (Exception e) {
	− LOG.fatal("Could not append. Requesting close of wal", e);
	+ LOG.warn("Could not append. Requesting close of wal", e);
	requestLogRoll();

```
// [caption=Wrong Log level]
@@
Logger log;
@@
...
− log.fatal(...);
+ log.warn(...);
```

Figure 4.15: Example SmPL patch corresponding to generic representation of the pattern associated to `FixMiner` pattern.

4.5.3 RQ3: Evaluation of Fix Patterns' Relevance for APR

Objective. We propose to assess whether fix patterns yielded by `FixMiner` are effective for automated program repair.

Protocol. We implement a prototype APR system that uses the fix patterns mined by `FixMiner` to generate patches for bugs by following the principles of the PAR [88], which is referred to as $\texttt{PAR}_{\texttt{FixMiner}}$ in the remainder of this work. In contrast with PAR where the templates were engineered by a manual investigation of example bug fixes, in $\texttt{PAR}_{\texttt{FixMiner}}$, the templates for repair are engineered based on fix patterns mined by `FixMiner`. Figure 4.16 overviews the workflow of $\texttt{PAR}_{\texttt{FixMiner}}$.

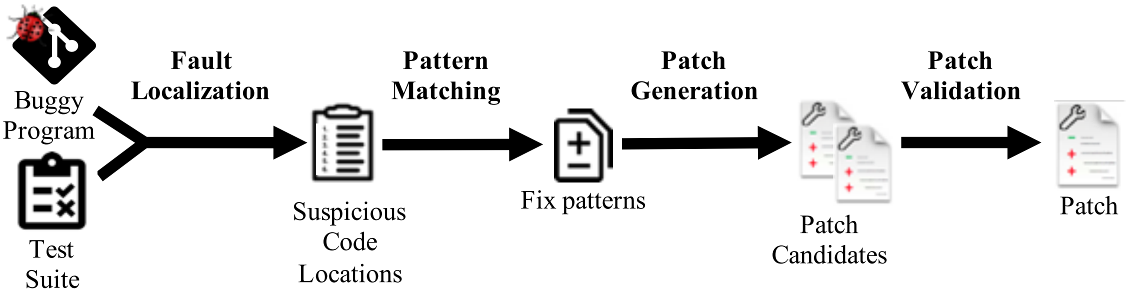


Figure 4.16: The overall workflow of $\texttt{PAR}_{\texttt{FixMiner}}$ program repair pipeline.

Fault Localisation. $\texttt{PAR}_{\texttt{FixMiner}}$ uses spectrum-based fault localisation. We use the GZoltar[7] [31] dynamic testing framework and leverage Ochiai [3] ranking metric to predict buggy statements based on execution coverage information of passing and failing test cases. This setting is widely used in the repair community [127, 152, 241, 253, 255], allowing for comparable assessment of $\texttt{PAR}_{\texttt{FixMiner}}$ against the state-of-the-art.

Pattern Matching and Patch Generation. Once the spectrum-based fault localisation (or information retrieval-based fault localisation [98, 242]) process yields a list of suspicious code locations, $\texttt{PAR}_{\texttt{FixMiner}}$ attempts to select fix patterns for each statement in the list. The selection of fix patterns is conducted

[7] We used GZoltar version 0.1.1

by matching the context information of suspicious code locations and fix patterns mined by `FixMiner`. Concretely, first, we parse the suspicious statement and traverse each node of its AST from its first child node to its last leaf node and form an AST subtree to represent its context. Then, we try to match the context (i.e., shape) of the AST subtree (from a suspicious statement) to the fix patterns' shapes.

If a matching fix pattern is found, we proceed with the generation of a patch candidate. Some fix patterns require donor code (i.e., source code extracted from the buggy program) to generate patch candidates with fix patterns. These are also often referred to as part of fix ingredients. Recall that, to integrate with repair tools, we leverage `FixMiner` Action patterns, which do not contain any code token information: they have "holes". Thus we search the donor code locally from the file which contains the suspicious statement. We select relevant donor code among the ones that are applicable to the fix pattern and the suspicious statement (i.e., data type(s) of variable(s), expression types, etc. that are matching to the context) to reduce the search space of donor code and further limit the generation of nonsensical patch candidates. For example, the fix pattern in Figure 4.17 can only be matched to a suspicious return statement that has a method invocation expression: thus, the suspicious return statement will be patched by replacing its method name with another one (i.e., donor code). The donor code is searched by identifying all method names from the suspicious file having the same return type and parameters with the suspicious statement. Finally, a patch candidate is generated by mutating suspicious statements with identified donor code following the actions indicated in the matched fix pattern. We generate as many patches as the number of identified pieces of donor code. Patches are generated consecutively in the order of matching within the AST.

Note: We remind the reader that in this study, we do not perform a specific patch prioritisation strategy. We search donor code from the AST tree of the local file that contains the suspicious statement by traversing each node of the AST of the local file from its first child node to its last leaf node in a breadth-first strategy (i.e., left-to-right and top-to-bottom). In case of multiple donor code options for a given fix pattern, the candidate patches are generated (each with a specific donor code) following the positions of donor codes in the AST tree (of the local file which contains the suspicious statement).

```
UPD ReturnStatement
———UPD MethodInvocation
——————UPD Simple@MethodName
```

Figure 4.17: Example of fix patterns yielded by `FixMiner`.

Pattern Validation. Once a patch candidate is generated, it is applied to the buggy program and will be validated against the test suite. If it can make the buggy program pass all test cases successfully, the patch candidate will be considered as a plausible patch, and $\texttt{PAR}_{\texttt{FixMiner}}$ stops trying other patch candidates for this bug. Otherwise, the pattern matching and patch generation steps are repeated until the entire suspicious code locations list is processed. Specifically, we consider only the first generated plausible patch for each bug to evaluate its correctness. For all plausible patches generated by $\texttt{PAR}_{\texttt{FixMiner}}$, we further manually check the equivalence between these patches and the oracle patch provided in Defects4J. If they are semantically similar to the developer-provided fix, we consider they as correct patches, otherwise remain as plausible.

Oracle. We use Defects4J[8] [78] dataset, which is widely used as a benchmark for Java-targeted APR research [35, 114, 150, 152]. The dataset contains 357 bugs with their corresponding developer fixes and test cases covering the bugs. Table 4.12 details statistics on the benchmark.

Results. Overall, we implemented the 31 fix patterns (i.e., Action patterns) mined by `FixMiner`, focusing only on the top-50 clusters (in terms of size).

[8] Version 1.2.0 - `https://github.com/rjust/defects4j/releases/tag/v1.2.0`

Table 4.12: Details of the benchmark.

Project	Bugs	LOC	Tests
JFreechart (Chart, C)	26	96K	2,205
Apache commons-lang (Lang, L)	65	22K	2,245
Apache commons-math (Math, M)	106	85K	3,602
Joda-Time (Time, T)	27	28K	4,130
Closure compiler (Closure, Cl)	133	90K	7,927
Total	357	321K	20,109

† In the table, column "Bugs" denotes the total number of bugs in Defects4J benchmark, column "LOC" denotes the number of thousands of lines of code, and column "Tests" denotes the total number of test cases for each project.

We compare the performance of $\texttt{PAR}_{\texttt{FixMiner}}$ against 13 state-of-the-art APR tools which have also used Defects4J benchmark for evaluating their repair performance. Table 4.13 illustrates the comparative results in terms of numbers of *plausible* (i.e., that passes all the test cases) and *correct* (i.e., that is eventually manually validated as semantically similar to the developer-provided fix) patches. Note that although HDRepair manuscript counts 23 bugs for which "correct" fixes are generated (and among which a correct fix is ranked number one for 13 bugs), the authors labelled fixes as "verified ok" for only 6 bugs (see artefact page [9]). We consider these 6 bugs in our comparison.

Overall, we find that $\texttt{PAR}_{\texttt{FixMiner}}$ successfully repaired 26 bugs from the Defects4J benchmark by generating correct patches. This performance is only surpassed to date by SimFix [76] that was concurrently developed with $\texttt{PAR}_{\texttt{FixMiner}}$.

Table 4.13: Number of bugs fixed by different APR tools.

Proj.	$\texttt{PAR}_{\texttt{FixMiner}}$	**kPAR**	**jGenProg**	**jKali**	**jMutRepair**	**Nopol**	**HDRepair**	**ACS**	**ssFix**	**ELIXIR**	**JAID**	**SketchFix**	**CapGen**	**SimFix**
Chart	**5/8**	3/10	0/7	0/6	1/4	1/6	0/2	2/2	3/7	4/7	2/4	6/8	4/4	4/8
Lang	**2/3**	1/8	0/0	0/	0/1	3/7	2/6	3/4	5/12	8/12	1/8	3/4	5/5	9/13
Math	**13/15**	7/18	5/18	1/14	2/11	1/21	4/7	12/16	10/26	12/19	1/8	7/8	12/16	14/26
Time	**1/1**	1/2	0/2	0/2	0/1	0/1	0/1	1/1	0/4	2/3	0/0	0/1	0/0	1/1
Closure	**5/5**	5/9	0/0	0/0	0/0	0/0	0/7	0/0	2/11	0/0	5/11	3/5	0/0	6/8
Total	**26/32**	17/47	5/27	1/22	3/17	5/35	6/23	18/23	20/60	**26/41**	9/31	19/26	21/25	**34/56**
P(%)	**81.3**	36.2	18.5	4.5	17.7	14.3	26.1	78.3	33.3	63.4	29.0	73.1	**84.0**	60.7

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (i.e., a patch that makes the program pass all test cases). Precision (P) means the precision of correctly fixed bugs in bugs fixed by each APR tool. kPAR [131] is the Java implementation of PAR. The data about jGenProg, jKali and Nopol are extracted from the experimental results reported by Martinez et al. [150]. The data of HDRepair [114] is collected from its author's reply. And the results of other tools are obtained from their papers in the literature (jMutRepair [152], ACS [255], ssFix [253], ELIXIR [200], JAID [35], SketchFix(SF) [72], CapGen [241] and SimFix [76]). The same for the data presented in Table 4.14.

Nevertheless, while these tools generate more correct patches than $\texttt{PAR}_{\texttt{FixMiner}}$, they also generate many more plausible patches which are however not correct. In order to comparatively assess the different tools, we resort to a Precision metric (P), which is the probability of correctness of the generated patches. P(%) is defined as the ratio of the number of bugs for which a correct fix is generated first (i.e., before any other plausible patch) against the number of bugs for which a plausible (but incorrect) patch is generated first. For example, 81% of $\texttt{PAR}_{\texttt{FixMiner}}$'s plausible patches are actually correct, while it is the case for 63% and 60% of respectively ELIXIR and SimFix plausible patches are correct. To date, only CapGen [241] achieves similar performance at yielding patches with slighter higher probability (at 84%) to be correct. The high performance of CapGen confirms their intuition that context-awareness, which we provide with `Rich Edit Script`, is essential for improving patch correctness.

Table 4.14 enumerates 128 bugs that are currently fixed (both correct and plausible) in the literature. 89 of them can be correctly fixed by at least one APR tool. $\texttt{PAR}_{\texttt{FixMiner}}$ generates correct patches for 26 bugs. Among the bugs in the used version of Defects4J benchmark, 267 bugs have not yet

[9] https://github.com/xuanbachle/bugfixes/blob/master/fixed.txt

been fixed by any tools in the literature, which still is a big challenge for automated program repair research.

Finally, we find that thanks to its automatically mined patterns, $\texttt{PAR}_{\texttt{FixMiner}}$ is able to fix six (6) bugs which have not been fixed by any state-of-the-art APR tools (cf. Figure 4.18).

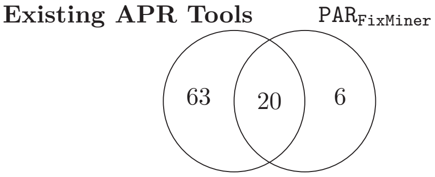


Figure 4.18: Overlap of the correct patches by $\texttt{PAR}_{\texttt{FixMiner}}$ and other APR tools.

RQ3▶ *Fix patterns (i.e., Action Patterns) yielded by* `FixMiner` *can be directly used in automated program repair pipelines and generates correct patches for buggy programs effectively. Additionally, the repair performance of* $\texttt{PAR}_{\texttt{FixMiner}}$*, which uses fix patterns yielded by* `FixMiner`*, is comparable to the state-of-the-art APR tools.*

Table 4.14: Defects4J bugs fixed by different APR tools.
"✓" indicates that the bug is correctly fixed, "✗" indicates the produced patch is plausible but not correct. "(✓)" indicates that a correct patch is generated by JAID, but is not the first plausible patch to be generated)".

Proj.	FixMiner	kPAR	SimFix	CapGen	SketchFix	JAID	ssFix	ACS	ELIXIR	HDRepair	jGenProg	jKali	jMutRepair	Nopol
C-1	✓	✓	✓	✓	✓	(✓)	✓		✓	✗	✗	✗	✓	
C-3		✗	✓						✗		✗			✗
C-4	✓	✓												
C-5		✗									✗	✗		✓
C-7		✓	✓								✗		✗	
C-8				✓	✓				✓	✗				
C-9					✓	(✓)			✓					
C-11	✓			✓	✓				✓					
C-12	✗	✗	✗											
C-13	✗	✗			✗				✗		✗	✗		✗
C-14			✗					✓						
C-15		✗									✗	✗		
C-17									✗					
C-18			✗											
C-19								✓						
C-20			✓		✓		✓							
C-21														✗
C-22			✗											
C-24	✓			✓	✓	✓	✓							
C-25	✗	✗									✗	✗	✗	✗
C-26	✓	✗			✗	✓						✗	✗	✗
Cl-2		✓												
Cl-5						✗								
Cl-10	✓									✗				
Cl-14			✓		✓		✓			✗				
Cl-18						✓								
Cl-21		✗												
Cl-22		✗												
Cl-31						(✓)								
Cl-33						✓								
Cl-38	✓	✓												
Cl-40						✓								
Cl-51										✗				
Cl-57			✓											
Cl-62	✓	✓	✓		✓	(✓)				✗				
Cl-63	✓	✓	✓			(✓)								
Cl-70					✗	✓				✗				
Cl-73	✓	✓	✓		✗	✓				✗				
Cl-79			✗											
Cl-106			✗											
Cl-109		✗												
Cl-115			✓				✓							
Cl-125						✗								
Cl-126		✗			✓	(✓)				✗				
L-6				✓	✓		✓		✓	✓				
L-7								✓						
L-10			✗							✗				
L-16			✓											
L-21							✓							
L-24						✗		✓	✓					
L-26				✓					✓					
L-27			✓										✗	
L-33			✓			✓	✓		✓					
L-35								✓						
L-38						(✓)			✓					
L-39			✓			✗		✗	✗					✗
L-41			✓											
L-43	✗	✗	✓	✓			✓		✓	✗				
L-44		✗	✗						✗					✓
L-45		✗	✗			(✓)								
L-46														✗
L-50			✓											
L-51		✗			✗	(✓)			✗	✓				✗
L-53		✗												✗
L-55					✓	(✓)								✓
L-57	✓	✗		✓					✓	✗				
L-58		✗	✓						✗					✓
L-59	✓	✓		✓	✓		✓		✓	✗				
L-60			✓											
L-61						✗								
L-63			✗											
M-1		✗	✗											
M-2									✗		✗	✗	✗	
M-3								✓						
M-4								✓						
M-5			✓	✓	✓	✓		✓	✓	✓	✓			
M-6			✗											
M-8		✗	✗								✗	✗		
M-10	✓													
M-15		✓												
M-20			✗						✗					
M-22	✓									✓				
M-25								✓						
M-28		✗	✗					✗			✗	✗	✗	
M-30	✓			✓			✓		✓					
M-32						(✓)			✗			✗		✗
M-33	✓	✓	✓	✓	✓		✓		✓					✗
M-34	✓								✓	✗				
M-35	✓		✓					✓						
M-40											✗	✗	✗	✗
M-41			✓				✓							
M-42														✗
M-49		✗									✗	✗		✗
M-50		✗	✓		✓	(✓)	✓		✓	✓	✓	✓	✗	✓
M-51		✗												
M-53			✓	✓		(✓)	✓			✓	✓			
M-57	✓		✓	✓			✓		✓				✗	✗
M-58	✓	✓		✓					✓				✗	✗
M-59			✓	✓	✓		✓		✓					
M-61								✓						
M-62		✗												
M-63		✗	✓	✓					✗					
M-65				✓										
M-69														✗
M-70	✓	✓	✓	✓	✓		✓		✓	✗	✓			
M-71			✓								✗			✗
M-72			✗											
M-73			✗		✗			✗	✗		✓			✗
M-75	✓	✓	✓	✓					✓					
M-78											✗	✗		✗
M-79	✓		✓	✗			✓							
M-80			✗	✗		(✓)	✓		✗		✗	✗		✗
M-81	✗	✗	✗	✗				✗			✗	✗	✗	✗
M-82	✓	✗	✗	✗	✓	(✓)		✓	✓	✗	✗	✗	✓	✗
M-84	✗	✗									✗	✗	✗	
M-85	✓	✓	✗	✓	✓	(✓)		✓	✓		✗	✗	✓	✗
M-87														✗
M-88			✗										✗	✗
M-89		✓						✓						
M-90								✓						
M-93								✓						
M-95											✗	✗		
M-97								✗						✗
M-98			✓											
M-99								✓						
M-104									✗					✗
M-105						✗								✗
T-4					✗				✓		✗	✗		
T-7		✓	✓											
T-11		✗							✗		✗	✗	✗	✗
T-15								✓	✓					
T-19	✓									✗				

4.6 Discussions and Threats to Validity

4.6.1 Runtime performance.

To run the experiments with `FixMiner`, we leveraged a computing system with 24 Intel Xeon E5-2680 v3 cores with 2.GHz per core and 3TB RAM. The construction of the `Rich Edit Scripts` took about 17 minutes. `Rich Edit Scripts` are cached in memory to reduce disk access during the computation of identical trees. Nevertheless, we recorded that comparing 1 108 060 pairs of trees took about 18 minutes.

4.6.2 Threats to external validity.

The selection of our bug-fix datasets carries some threats to external validity that we have limited by considering known projects, and heuristics used in previous studies. We also make our best effort to link commits with bug reports as tagged by developers. Some false positives may be included if one considers a strict and formal definition of what constitutes a bug.

4.6.3 Threats to construct validity

arise when checking the compatibility of `FixMiner`'s patterns against the patterns used by literature APR systems. Indeed, for the comparison, we do not conduct exact mapping where the elements should be the same, given that literature patterns can be more abstract than the ones yielded by `FixMiner`. For example, *Modify Method Name* (i.e., FP10.1) is a sub-fix pattern of *Mutate Method Invocation Expression* (i.e., FP10), which is about replacing the method name of a method invocation expression with another appropriate method name [133]. This fix pattern can be matched to any statement that contains a method name under method invocation expression. However, in this work, the similar fix patterns yielded by `FixMiner` have more context information. Therefore, we consider context information to check the compatibility of `FixMiner`'s patterns against the patterns used by literature APR systems. For example, the fix pattern shown in Figure 4.17 is to modify the buggy method name of a method invocation expression with another appropriate method name which is inside a `Return-Statement`. As the context information refers to a `Return-Statement` the fix pattern shown in Figure 4.17 considered as compatible with *Mutate Return Statement* (i.e., FP12.). Nevertheless, the mapping is conservative in the sense that we consider that a `FixMiner` pattern matches a pattern from the literature as long as it can fit with the literature pattern.

4.7 Related Work

4.7.1 Automated Program Repair.

Patch generation is one of the key tasks in software maintenance since it is time-consuming and tedious. If this task is automated, the cost and time of developers for maintenance will be dramatically reduced. To address the issue, many automated techniques have been proposed for program repair [163]. GenProg [117], which leverages genetic programming, is a pioneering work on program repair. It relies on mutation operators that insert, replace, or delete code elements. Although these mutations can create a limited number of variants, GenProg could fix several bugs (in their evaluation, test cases were passed for 55 out of 105 real program bugs) automatically, although most of them have been found to be incorrect patches later. PACHIKA [41] leverages object behaviour models. SYDIT [158] and LASE [159] automatically extracts an edit script from a program change. While several techniques have focused on fixability, Kim et al. [88] pointed out that patch acceptability should be considered as well in program repair. Automatically generated patches often have nonsensical structures and logic even though those patches can fix program bugs with respect to program behaviour (i.e., w.r.t. test cases). To address this issue, they proposed PAR, which leverages manually-crafted fix patterns. Similarly, Long and Rinard proposed Prophet [140] and Genesis [137], which generates patches by leveraging fix patterns extracted from the history of changes in repositories. Recently, several approaches [17, 63] leveraging deep learning have been proposed for learning to fix bugs. Even recent APR approaches that target bug reports rely on fix templates to generate patches. iFixR [100] is such an example which builds on top of the templates built TBar [133] templates. Overall, we note that the community is going in the direction of implementing repair strategies based on fix patterns or templates. Our work is thus essential in this direction as it provides a scalable, accurate and actionable tool to mine such relevant patterns for automated program repair.

4.7.2 Code differencing.

Code differencing is an important research and practice concern in software engineering. Although commonly used by human developers in manual tasks, differencing at the text line-level granularity [167] is generally unsuitable for automated analysis of changes and the associated semantics. AST differencing work has benefited in the last decade for the extensive investigations that the research community has performed for general tree differencing [8, 18, 33, 37]. ChangeDistiller [56] and GumTree [52] constitute the current state-of-the-art for AST differencing in Java. In this work, we have selected GumTree as the base tool for the computation of edit scripts as its results have been validated by humans, and it has been shown to be more accurate and fine-grained edit scripts. Nevertheless, we have further enhanced the edit script yielding an algorithm that keeps track of contextual information. Our approach echoes a recently published work by Huang et al. [73]: their CLDIFF tool similarly enriches the AST produced by GumTree to enable the generation of concise code differences. The tool however was not available at the time of our experiments. Thus, to satisfy the input requirements of our fix pattern mining approach, we implement `Rich Edit Script`, to enrich GumTree-yielded edit scripts by retaining more contextual information.

4.7.3 Change patterns.

The literature includes a large body of work on mining change patterns.

4.7.3.1 Mining-based approaches.

In recent years, several approaches have built upon the idea of mining patterns or leveraging templates. Fluri et al., based on edit scripts computed by their ChangeDistiller AST difference, have used hierarchical clustering to discover unknown change types in three Java applications [55]. They have limited themselves however to considering only changes implementing the 41 basic change types that they had previously identified [54]. Kreutzer et al. have developed C3 to automatically detect groups of similar code changes in code repositories with the help of clustering algorithms [102]. Martinez and Monperrus [151] assessed the relationship between the types of bug fixes and automatic program repair. They perform extensive large scale empirical investigations on the nature of human bug fixes based on fine-grained abstract syntax tree differences by ChangeDistiller. Their experiments show that the mined models are more effective for driving the search compared to random search. Their models however remain at a high level and may not carry any actionable patterns to be used by other template-based APR. Our work however also targets systematising and automating the "mining of actionable fix patterns" to feed pattern-based program repair tools.

An example application is related to work by Livshits and Zimmermann [136] who discovered application-specific repair templates by using association rule mining on two Java projects. More recently, Hanam et al. [64] have developed the BugAID technique for discovering most prevalent repair templates in JavaScript. They use AST differencing and unsupervised learning algorithms. Our objective is similar to theirs, focusing on Java programs with different abstraction levels of the patterns. `FixMiner` builds on a three-fold clustering strategy where we iteratively discover recurrent changes preserving surrounding code context.

4.7.3.2 Studies on code change redundancies.

A number of empirical studies have confirmed that code changes are repeatedly performed in software code bases [91, 93, 161, 272]. Same changes are prevalent because multiple occurrences of the same bug require the same change. Similarly, when an API evolves, or when migrating to a new library/framework, all calling code must be adopted by the same collateral changes [177]. Finally, code refactoring or routine code cleaning can lead to similar changes. In a manual investigation, Pan et al. [181] have identified 27 extractable repair templates for Java software. Among other findings, they observed that if-condition changes are the most frequently applied to fix bugs. Their study, however, does not discuss whether most bugs are related to If-condition or not. This is important as it clarifies the context to perform if-related changes. Recently, Nguyen et al. [173] have empirically found that 17-45% of bug fixes are recurring. Our focus is to provide tool-support automated approach to inferring change patterns in a dataset to drive repair patterns to guide APR mutation. Moreover, our patterns are less generic than the ones in previous works (e.g., as in [173, 181]).

Concurrently to our work, Jiang et al. have proposed SimFix [76], and Wen et al. CapGen [241] which implements a similar idea of leveraging code redundancies using contextual information for shaping the program repair space. In `FixMiner` however, the pattern mining phase is independent of the patch generation phase, and the resulting patterns are tractable and reusable as input to other APR systems.

4.7.3.3 Generic and semantic patch inference.

Ideally, `FixMiner` is a tool that aims at performing towards finding a generic patch that can be leveraged by automated program repair to correctly update a collection of buggy code fragments. This problem has been recently studied by approaches such as `spdiff` [9, 10] which work on the inference of generic and semantic patches. This approach, however, is known to be poorly scalable and has constraints of producing ready-to-use semantic patches that can be used by the Coccinelle matching and transformation engine [28]. There have however a number of prior works that tries to detect and summarise program changes. A seminal work by Chawathe et al. describes a method to detect changes to structured information based on an ordered tree and its updated version [33]. The goal was to derive a compact description of the changes with the notion of minimum cost edit script which has been used in the recent ChangeDistiller and GumTree tools. The representations of edit operations, however, are either often to overfit to a particular code change or abstract very loosely the change so that it cannot be easily instantiated. Neamtiu et al. [169] proposed an approach for identifying changes, additions and deletions of C program elements based on the structural matching of syntax trees. Two trees that are structurally identical but have differences in their nodes are considered to represent matching program fragments. Kim et al. [92] have later proposed a method to infer "change-rules" that capture many changes. They generally express changes related to program headers (method headers, class names, package names, etc.). Weissgerber et al. [239] have also proposed a technique to identify likely refactorings in the changes that have been performed in Java programs. Overall, these generic patch inference approaches address the challenges of how the patterns that will be leveraged in practice. Our work goes in that direction by yielding different kinds of patterns for different purposes: shape-based patterns reduce the context of code to match; action patterns are the ones that correspond to fix patterns used in the repair community; token patterns are used for inferring collateral evolutions.

4.8 Summary

We have presented FixMiner, a systematic and automated approach to mine relevant and actionable fix patterns for automated program repair. The approach builds on an iterative and three-fold clustering strategy, where in each round forming clusters of identical trees representing recurrent patterns.

We have evaluated FixMiner on thousands of software patches collected from open source projects. Preliminary results show that we are able to mine accurate patterns, efficiently exploiting change information in Rich Edit Scripts. We assess the consistency of the mined patterns with the patterns in the literature. We further integrated the mined patterns to an automated program repair prototype, $\texttt{PAR}_{\texttt{FixMiner}}$, with which we are able to correctly fix 26 bugs of the Defects4J benchmark. Beyond this quantitative performance, we show that the mined fix patterns are sufficiently relevant to produce patches with a high probability of correctness: 81% of $\texttt{PAR}_{\texttt{FixMiner}}$'s generated plausible patches are correct.

5 Analysing Communication Channels

Bug tracking is now commonplace in software development ecosystems. Development teams in large-scale systems set up dedicated systems (e.g., Linux) and smaller software projects alike [21]. Bug tracking systems (such as Bugzilla[1] and Jira[2]) implement a communication channel between developers and software users, and are used by developers themselves to keep track of the bugs that they encounter. Bugs are indeed reported in natural language, where users tentatively describe the execution scenario that was being carried out and the unexpected outcome (e.g., crash stack traces). Such bug reports constitute an essential artefact within a software development cycle and can become an overwhelming concern for maintainers. Thus, an ultimate automation target of software maintenance is then the systematisation of patch generation for user-reported bugs.

Bug localisation is such a typical task, where text in a bug report is analysed to identify file locations in the source code that can be associated with the reported bug. Many automated tasks in software maintenance rely on information retrieval (IR) techniques to identify specific information within unstructured data. Unfortunately, despite the promising results reported in the literature, IR-based bug localisation tools are still not adopted in practice. We argue that one reason could be that researchers build "one-size-fits-all" approaches, without fully addressing the differences of available information that may exist across bug reports.

In this work, first, we extensively study the performance of state-of-the-art bug localisation tools, specifically focusing on investigating the query formulation (i.e., which bug report features should be compared against which features of source code files) and its importance with respect to the localisation performance. Building on insights from this study, we propose a new learning approach where multiple classifier models are trained on clear-cut sets of bug-location pairs. Concretely, we apply a gradient boosting supervised learning approach to various sets of bug reports whose localisations appear to be successful with specific types of features. The training scenario builds on our findings that the various state-of-the-art localisation tools (hence the associated similarity features that they leverage) can be highly performant for specific sets of bug reports. We implement `D&C`, a multi-classifier approach, which computes appropriate weights that should be assigned to the similarity measurements between pairs of information token types (the bug report and source code).

The APR literature has, so far, mostly focused on *generate-and-validate* setups where a well-defined test suite drives fault localisation and patch generation. On the one hand, however, the common (yet strong) assumption on the existence of relevant test cases does not hold in practice for most development settings: many bugs are reported without the available test suite being able to reveal them. On the other hand, for many projects, the number of bug reports generally outstrips the resources available to triage them. Towards increasing the adoption of patch generation tools by practitioners, we investigate a new repair pipeline, `iFixR`, driven by bug reports: (1) bug reports are fed to an IR-based fault localiser; (2) patches are generated from fix patterns and validated via regression testing; (3) a prioritised list of generated patches is proposed to developers.

This chapter is based on the works published in the following research papers:

[1] `https://www.bugzilla.org`

[2] `https://jira.atlassian.com`

- A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv preprint arXiv:1902.02703*, 2019
- A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon. 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*, pages 314–325. ACM, 2019

5.1 Learning Insights from Bug Reports

5.1.1 Overview

A typical bug report is a natural language description of a problem that a user has encountered while interacting with the software product, including more or less details on how to reproduce the bug. This report may further include other structural information such as the stack trace that was produced during a crashed execution. To fix the reported bug, developers must analyse it and eventually locate the relevant buggy code.

Automated bug localisation aims at reducing the developer effort and the time cost in the manual inspection of source code when attempting to identify relevant buggy code files or functions. Automation is particularly essential for large software projects which are flooded by bug reports that must each be mapped to a relevant source code file among the thousands forming the software project. To address the challenge of automating localisation, the research community has recently investigated various information retrieval (IR) techniques [25, 42, 58, 146, 201, 205]. In the proposed tools [143, 199, 232, 242, 247, 267, 276], information tokens are extracted from a given bug report to formulate a query to be matched in a search space of documents formed by the collections of source code files and indexed through tokens extracted from source code properties. IR-based bug localisation (IRBL) tools then rank the documents based on a probability of relevance (often measured as a similarity score). Highly ranked files are predicted to be the ones that are likely to contain the buggy code. This process is thus expected to reduce the number of files on which a developer must focus her examination.

Despite growing interest in the literature, with numerous approaches continuously claiming new performance improvements over the state-of-the-art, we are not aware of any adoption in the developer community, nor any integration in other research approaches such as automated repair. As demonstrated by a recent study by Lee et al. [120], this is mainly due to:

1. *A limited performance from the state-of-the-art*: to date, empirical assessments in the literature [120] indicate Mean Average Precision metrics between 0.35 and 0.38 and Mean Reciprocal Rank metrics between 0.43 and 0.52. Concretely, even the best performing IRBL tools still fail every other time to adequately associate the bug reports with the relevant source code files.
2. *A lack of comprehensive validation of the value of the various IR features*[3]: literature approaches incrementally add new dimensions of comparison by considering additional information features or by changing the weight of different information tokens. Unfortunately, to date, the community still lacks a clear overview of the information gain that each feature retrieved through IR contributes to the localisation process.

Towards contributing to pushing the frontiers of IR-based bug localisation (IRBL) further, we propose to undertake a comprehensive investigation on the information gain provided by a variety of features that are commonly extracted from bug reports and source files for IRBL tasks. By further correlating the use of specific feature sets with the performance of different state-of-the-art tools, we are able to establish the need for building an IRBL approach where the weight of similarity scores between bug report features and source code features are learned for different specific groupings of bug reports. We refer to it as a "divide-and-conquer" (`D&C`) strategy, which offers an opportunity to improve the overall performance of IRBL in large-scale experiments substantially.

In a typical IRBL tool, given a bug report and the source code files of a project, the weights associated to the similarity scores between extracted features are statically set and will be the same for all bug reports/source code files. In contrast, `D&C` aims to dynamically select the weights that must be used for a given pair (bug report/source code file) following a training phase which learns what kinds

[3] Generally, features are derived from tokens. As a result, for convenience the terms features and tokens are used interchangeably in this chapter.

of information tokens are important to these "kinds" of bug reports/files pairs. We approximate the "kind" of bug reports/file pairs by building on training sets that are successfully and exclusively localised by existing state-of-the-art tools. In summary, this work presents the following main contributions to the research community on IRBL:

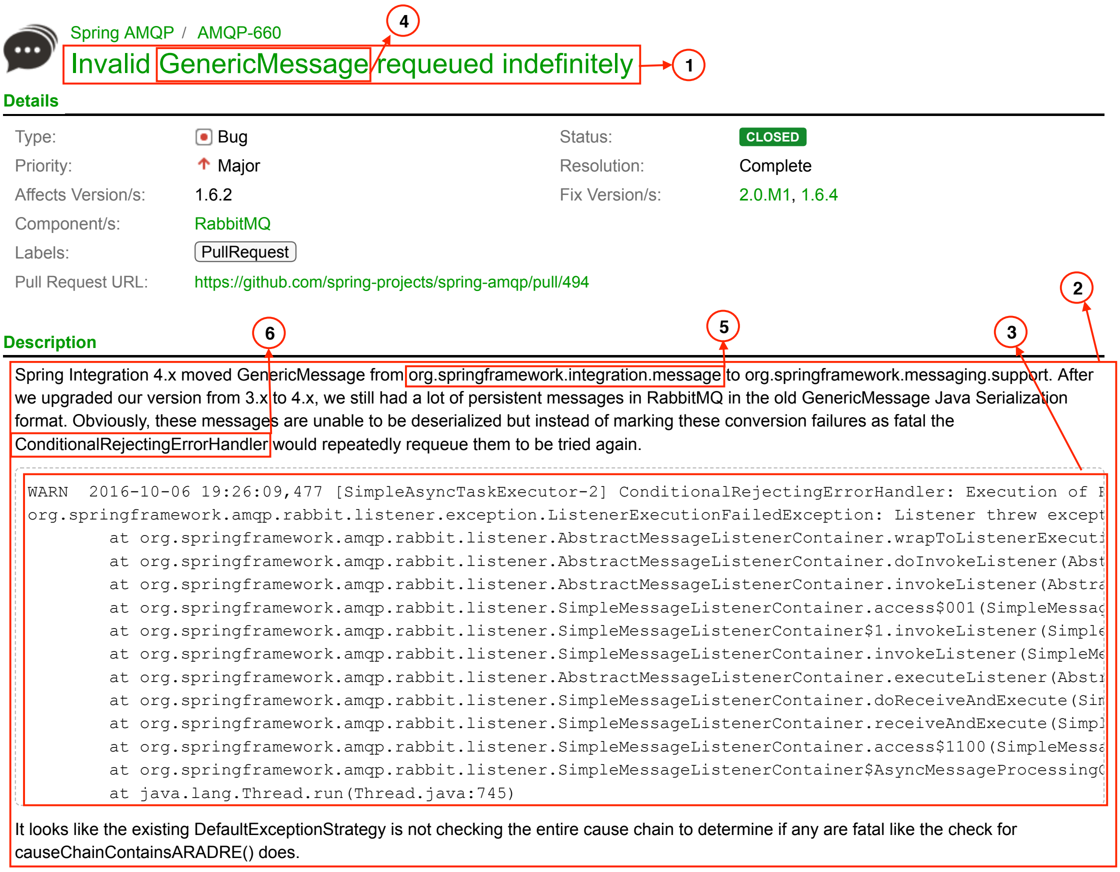
- We dissect the query formulation (*i.e., what information tokens from bug reports are used to search for relevant buggy files by matching appropriate information tokens from source code*) in state-of-the-art IRBL systems. Then we assess the contribution of different information tokens on the localisation performance. Our experiments compare six state-of-the-art IRBL tools leveraging an extensive database collected for Bench4BL [120]. We observe that different groups of bug reports that have been successfully localised by all or at least one or only one state-of-the-art tool have distinctive attributes which make specific information tokens more or less significant, in terms of contribution to the localisation performance.
- We propose a learning approach to improve the performance of IRBL. The key idea of the approach (named `D&C`) is based on the finding that each state-of-the-art tool appears to be providing good localisation performance on a specific set of bug reports where others fail. The gradient boosting learning algorithm is leveraged to capture the weight that the different information tokens have when the association of a bug report with a source code file, from a specific subset of the dataset, is a successful localisation. We then build a multi-classifier where prediction probabilities by different classifiers are combined and reordered to yield the IRBL ranked list of potentially buggy files.
- We extensively assess `D&C` using Bench4BL [120], the largest and most comprehensive benchmark that was recently proposed in the literature. The data are from 45 projects (amounting to 5 321 bug reports and 70 675 java files). We empirically show that the proposed `D&C` approach outperforms the state-of-the-art to yield record Mean Average Precision and Mean Reciprocal Rank values for IRBL, respectively at 0.52 and 0.63. `D&C` is further able to localise 50% of bugs at Top1, 77% at Top5 and 85% at Top10.

The remainder of this chapter is organised as follows: we first detail background information on tools used and features leveraged in IR-based localisation in Section 5.2. The empirical study for dissecting IRBL performance is described in Section 5.3. Our `D&C` approach is presented in Section 6.3 and evaluated in Section 5.5. We provide final remarks about our work in Section 5.6, discuss related work in Section 5.7 and conclude in Section 5.8.

5.2 Background

In the context of IR-based bug localisation (IRBL), each bug report is treated as a query while the source files in a project form a document collection (i.e., the target search space). Since the performance of IR systems is generally limited by the linguistic variations present in natural language texts [205], a classic IR challenge lies in effectively recognising the features in the query and document [58, 61]. Towards providing state-of-the-art tools for IRBL, researchers have investigated a variety of information tokens that can be identified in bug reports and source code files. We conducted a quick literature review in order to identify which features (i.e., information tokens) are considered by state-of-the-art tools. We consider the state-of-the-art tools that have been studied by Lee et al. [120] in a recent comprehensive reproduction study.

A bug report, such as the one illustrated in Figure 5.1, is generally submitted after encountering an issue while running a software program, and typically provides a description of a failure. It is then stored in a bug tracking system for investigation by project developers. The bug tracking system then records the time at which the bug was reported, the identity of the person who reported it, as well as other information related to the severity or the affected software version. Occasionally, the bug description may include information on the erroneous program behaviour, and the details on how to reproduce the bug hint at the location of the fault in the code (in the form of code blocks or stack traces). These information provided with the bug reports can be processed to extract relevant features that could be relevant for implementing IRBL. For example, code-related terms such as package names and class names found in the summary and description, in addition to stack traces and code blocks, as separate features referred to as *hints*.



Spring AMQP / AMQP-660

Invalid GenericMessage requeued indefinitely

Details

Type:	Bug	Status:	CLOSED
Priority:	Major	Resolution:	Complete
Affects Version/s:	1.6.2	Fix Version/s:	2.0.M1, 1.6.4
Component/s:	RabbitMQ		
Labels:	PullRequest		
Pull Request URL:	https://github.com/spring-projects/spring-amqp/pull/494		

Description

Spring Integration 4.x moved GenericMessage from org.springframework.integration.message to org.springframework.messaging.support. After we upgraded our version from 3.x to 4.x, we still had a lot of persistent messages in RabbitMQ in the old GenericMessage Java Serialization format. Obviously, these messages are unable to be deserialized but instead of marking these conversion failures as fatal the ConditionalRejectingErrorHandler would repeatedly requeue them to be tried again.

```
WARN  2016-10-06 19:26:09,477 [SimpleAsyncTaskExecutor-2] ConditionalRejectingErrorHandler: Execution of R
org.springframework.amqp.rabbit.listener.exception.ListenerExecutionFailedException: Listener threw except
        at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer.wrapToListenerExecuti
        at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer.doInvokeListener(Abst
        at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer.invokeListener(Abstra
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.access$001(SimpleMessag
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer$1.invokeListener(Simple
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.invokeListener(SimpleMe
        at org.springframework.amqp.rabbit.listener.AbstractMessageListenerContainer.executeListener(Abstr
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.doReceiveAndExecute(Sim
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.receiveAndExecute(Simpl
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer.access$1100(SimpleMessa
        at org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer$AsyncMessageProcessingC
        at java.lang.Thread.run(Thread.java:745)
```

It looks like the existing DefaultExceptionStrategy is not checking the entire cause chain to determine if any are fatal like the check for causeChainContainsARADRE() does.

Figure 5.1: Example bug report with (1) Summary, (2) Description, (3) Stack Trace, (4) Summary Hint, (5) Description Hint, and (6) Code Element.

In general, state-of-the-art tools to IRBL develop specific strategies towards ensuring that queries are processed adequately to find the information that allows accurate matching with source code file information. Nevertheless, a key impactful factor in the performance of the approach remains the features that are extracted as representative and discriminating information of bug reports and source code files. In order to identify which features are considered in the literature, we provide the following summary information for recent tools:

- Zhou et al. [276] have initiated the breakthrough in IRBL by radically raising the precision to about 50%. The tool merely treats source code as text to match with natural language text of bug reports. Moreover, it leverages the similarity among bug reports to guide localisation, and uses file sizes to weight probability scores (given that larger files are more likely to include bugs).
- Saha et al. [199] proposed to treat separately summary and description parts of a bug report. They further extracted specific information from source code files into a structured format (class names, variable names, comments) to improve matching.
- Wang et al. [232] combined the works of Zhou et al. [276] and Saha et al. [199] and further considered version history to improve prediction (a previously buggy file is likely to contain bugs). They later extended their work to consider reported information [233].
- Wong et al. [247] proposed to segment source code files into smaller segments and used stack trace information to improve bug localisation.

- Youm et al. [267], in addition to information tokens such as stack traces, method names, and similarity among bug reports and method names, further consider method level matching and exploit comments from bug reports.
- Wen et al. [242] have focused on the change level in source code and attempted to separate natural language tokens from code entities to improve matching between the bug report and source code.

In this work, we consider extracting such common features which can be reliably and readily computed in a large scale dataset (e.g., we do not consider similarity among bug reports because of the combinatorial explosion of pairwise combinations), and which are available once a bug report is submitted (e.g., we do not consider comments which may be subsequently added to the bug reports). Overall, we focus in this study on :

1. *Textual information of bug reports and source code files*: source code files that are textually similar to the bug report text tend to be associated with the reported bug [143].
2. *Structured information from source code*: different fields in code (e.g., class names, package names, comment, etc.) have varying importance for matching bug report vocabulary [165, 199].
3. *Structured information from bug reports*: different parts of bug reports, such as title and body, may contain specific or verbose information for matching. Furthermore, some code elements can often be identified in bug reports, which could be more effective for bug localisation [199, 247, 267].
4. *Stack traces*: the bug location is likely among the classes or methods listed in the stack trace [203, 247].
5. *Segmentation*: matching at the code hunk level [242] or dividing source code files into equally sized segments [247] can provide more accuracy in localizing bugs [262].
6. *Commit log*: messages included in source code version management systems can provide the description of functionalities that match user bug report text better than source code tokens [242].

The detailed list of features is presented in Section 5.3.6.

5.3 Empirical Study on IRBL tools

In this section, we describe the setup and results of a large empirical study that we have conducted to investigate the impact of different IRBL features as well as the differences in performances by current state-of-the-art tools. Our objectives are to assess the impact of the query formulation on the performance, and to provide comprehensive insights into the value of different IR features for bug localisation. We recall that this study is focused on working tools available to the community. We refer the reader to the study of Thomas et al. [223] on the impact of classifier configuration, which focuses on the underlying IR methods.

5.3.1 Research Questions

Our study focuses on the following questions:

RQ-1: *Are state-of-the-art tools diversely successful depending on the samples of the benchmark?* A recent reproducibility study has shown that current tools have an overall similar performance in terms of average precision [120]. However, the authors did not assess with the large dataset of Bench4BL [120] whether these tools have affinities for specific sets of bug reports/code files.
RQ-2: *Which combinations of features provide the best information gain for IR-based bug localisation?* Although the literature recurrently adds new features that are expected to improve overall localisation, little knowledge has been established by the community on the actual contribution of each feature, and whether this contribution varies depending on the project.

5.3.2 Experiment Setup

For the purpose of our study, we consider six state-of-the-art tools which are broadly used in the literature. Although some works have been later extended by tweaking some parameters and features, we consider the originally-published work and the associated implementation details to perform our study. Table 5.1 enumerates the tools of which the implementations were readily available and have been applied to the Bench4BL benchmark by Lee et al. [120].

Table 5.1: Tools considered in this study.

Name	Venue	Year
BugLocator [276]	Intl. Conf. on Software Engineering	2012
Bluir [199]	Intl. Conf. on Automated Software Engineering	2013
Amalgam [232]	Intl. Conf. on Program Comprehension	2014
Brtracer [247]	Intl. Conf. on Software Maintenance and Evolution	2014
Blia [267]	Asia-Pacific Software Engineering Conference	2015
Locus [242]	Intl. Conf. on Automated Software Engineering	2016

5.3.3 Dataset

To conduct our study, we exploit the dataset and benchmark provided by Bench4BL [120]. This benchmark was recently proposed by Lee et al. in an effort to push the assessment of current tools. Bench4BL (i.e., a benchmark for Bug localisation) was then leveraged to perform a comprehensive reproduction study on state-of-the-art IRBL tools. Table 6.1 enumerates the projects available in the Bench4BL benchmark. For the purpose of our study, we have thoroughly investigated the datasets and applied further constraints to obtain a clean dataset.

For each dataset, we have performed an extra curation step by ensuring that all files tagged as fixing a given bug are still available in the latest code version of the Git repository (as of December 2018). When at least one of these files are not available, we discard the associated bug report from our experiments. Eventually, our experiments are done on 5 321 bug reports filed in 45 projects (whereas BenchBL originally includes 8 652 reports from 46 projects).

In order to identify bug-fixing patches, Bench4BL leverages the bug linking strategies enforced when developers use the JIRA bug tracking system. Bug tracking systems are crawled, and bug links are verified based on two checks: i) Bench4BL checked for explicit commit ids (i.e., Git hashes) and file paths associated to the bug on the bug tracking database: for each file impacted by an identified commit, it considers the corresponding change as a bug fix change. ii) Similarly, Bench4BL also checked commit logs to identify bug report ID and associate the corresponding changes as bug fix changes. Finally, the Bench4BL dataset is curated by selecting only bug reports that are indeed considered as such and are thus resolved and tagged as RESOLVED or FIXED, and completed with status CLOSED. Eventually, the *cleaned Bug reports* amount to 5321 (second column of Table 6.1).

For the assessment of our proposed approach[4], we further clean the data from all bug reports that are suspected of being post-fix activities. We consider such bug reports to represent future data and may thus lead to artificial performance. For example, some bug reports are submitted by developers to keep track of what the code changes are meant to correct. The associated descriptions are too precise and may unrealistically match the source code (e.g., with file names and method names) with very high accuracy. Concretely we dismiss cases where the bug reporter and bug fixer (change committer) are same. This equality is controlled via the email. We also remove bug reports cases where a patch attachment is provided by the reporter or via a comment within the hour. Eventually,

[4] The empirical study part is done with all bug reports as we are blindly investigating the importance of bug report features (whether future or past data)

Table 5.2: Descriptive Statistics of Curated Bench4BL.

Project	# Cleaned Bug Reports	# with Same Email	# with Attachment	# Pre-fix Bug Reports	# Source Code Files	# Bug Report-Source
APACHE-CAMEL	1114	128	182	797	18671	20799494
APACHE-CODEC	39	5	7	28	126	4914
APACHE-COLLECTIONS	32	0	5	26	535	17120
APACHE-COMPRESS	99	16	28	56	354	35046
APACHE-CONFIGURATION	13	0	1	11	458	5954
APACHE-CRYPTO	1	0	0	1	87	87
APACHE-CSV	12	1	3	8	31	372
APACHE-HBASE	452	73	168	228	3758	1698616
APACHE-HIVE	727	156	234	395	6200	4507400
APACHE-IO	78	12	10	43	246	19188
APACHE-LANG	144	31	14	99	324	46656
APACHE-MATH	17	4	1	4	1324	22508
APACHE-WEAVER	2	1	0	1	89	178
JBOSS-ELY	19	0	0	5	936	17784
JBOSS-ENTESB	6	0	0	6	23	138
JBOSS-JBMETA	11	0	1	10	852	9372
JBOSS-SWARM	38	0	2	35	1358	51604
JBOSS-WFARQ	1	0	0	1	171	171
JBOSS-WFCORE	321	0	11	310	4141	1329261
JBOSS-WFLY	519	0	55	461	8581	4453539
JBOSS-WFMP	3	0	0	3	84	252
SPRING-AMQP	77	7	3	67	453	34881
SPRING-ANDROID	7	0	0	7	305	2135
SPRING-BATCH	253	1	27	239	1897	479941
SPRING-BATCHADM	15	0	0	15	231	3465
SPRING-DATACMNS	119	0	8	110	745	88655
SPRING-DATAGRAPH	3	1	0	2	287	861
SPRING-DATAJPA	119	0	16	103	367	43673
SPRING-DATAMONGO	213	0	17	195	821	174873
SPRING-DATAREDIS	41	3	2	37	720	29520
SPRING-DATAREST	77	1	2	72	419	32263
SPRING-LDAP	42	2	6	29	529	22218
SPRING-MOBILE	11	0	0	11	105	1155
SPRING-ROO	130	0	17	113	1077	140010
SPRING-SEC	251	0	20	242	2304	578304
SPRING-SECOAUTH	18	3	1	14	760	13680
SPRING-SGF	30	8	5	19	902	27060
SPRING-SHDP	40	26	0	14	1068	42720
SPRING-SOCIAL	13	0	0	35	217	2821
SPRING-SOCIALFB	12	1	0	11	261	3132
SPRING-SOCIALLI	4	0	0	4	183	732
SPRING-SOCIALTW	8	0	1	7	156	1248
SPRING-SPR	31	0	5	25	6906	214086
SPRING-SWF	53	0	7	40	748	39644
SPRING-SWS	106	1	10	66	865	91690
Total	5321	481	869	4005	70675	35088421

the *pre-fix bug reports* amount to 4005 (fifth column of Table 6.1) Table 6.1 reports all the statistics of the datasets. In the end, our experiments are done with similarity computations for over 35 million bug report-source code file pairs.

5.3.4 Performance Metrics

Assessment is quantified with common metrics used in the literature, with a focus on Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR).

Precision is a measure of accuracy in bug localisation that shows how many files are correctly recommended within given TopN files.

$$\mathrm{P}(n) = \frac{\#of\ buggy\ files\ at\ top\ n}{n} \tag{5.1}$$

Recall is a measure of coverage in bug localisation that shows how many files are correctly recommended within given TopN files over the actually fixed files by a developer for a given bug report.

$$\mathrm{R}(n) = \frac{\#of\ buggy\ files\ at\ top\ n}{\#of\ actual\ fixed\ files} \tag{5.2}$$

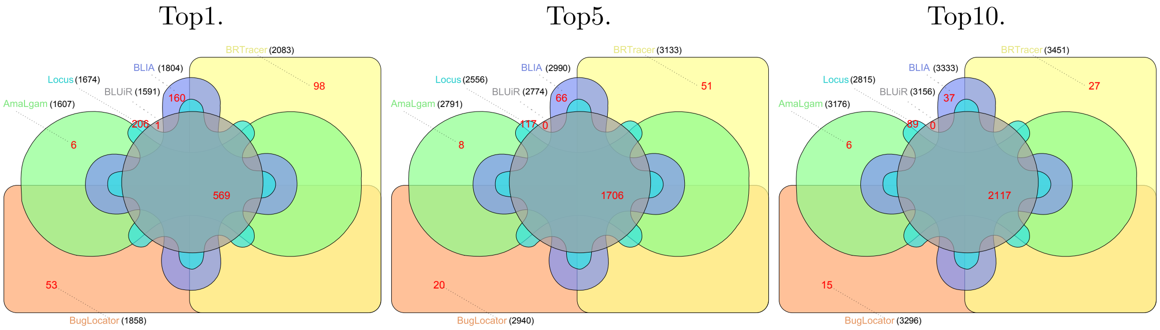


Figure 5.2: Successful recommendations of IRBL tools and the overlapping among them for Top1, Top5 and Top10.

Average Precision is computed for a given bug report by aggregating precision values of several positively recommended files.

$$AP = \sum_{i=1}^{N} = \frac{\mathrm{P}(i) \times \mathrm{pos}(i)}{\#of\ positive\ instances} \tag{5.3}$$

where N is the number of ranked files by an IRBL tool, i is a rank in the ranked list of recommended files. $\mathrm{pos}(i)$ indicates whether the i_{th} file in the ranked list is a buggy file (i.e., $\mathrm{pos}(i) \in 0, 1$).

- **Mean Average Precision (MAP)** is computed by taking the mean value of AP values across all bug reports:

$$MAP = \frac{1}{M} \sum_{j=1}^{M} \mathrm{AP}(j) \tag{5.4}$$

where M is the number of all bug reports and $\mathrm{AP}(j)$ is the average precision of bug report j.

- **Mean Reciprocal Rank (MRR)** computes the mean value of the position of the first buggy file in the ranked list recommended an IRBL tool as follows:

$$MRR = \frac{1}{M} \sum_{j=1}^{M} \frac{1}{rank_j} \tag{5.5}$$

where M is the number of all bug reports and $rank_i$ means the position of the first buggy file in the ranked list for i_{th} bug report.

- **Top-N Rank (TopN)** computes the number of bug reports having at least one relevant file in the first N files of the retrieved list.

$$TopN(r) = \begin{cases} 1 & \begin{array}{l}\text{report } r \text{ having at least} \\ \text{one relevant file in the within} \\ \text{top-}N \text{ files recommended}\end{array} \\ 0 & \text{otherwise} \end{cases} \tag{5.6}$$

$$TopN = \sum_{r=1}^{R} TopN(r) \tag{5.7}$$

where $r \in R$ is a bug report, and $N \in \mathbb{N}$ is a parameter of how many recommendations (i.e., files to fix) to look up from the results of f. Let R be a set of bug reports, and $\mathbb{N}$ 1, 5, 10 respectively. We also present a version of this metric in terms of percentage against the total number of bug reports that must be validated.

5.3.5 RQ-1: Affinities among state-of-the-art tools

To answer our first research question (RQ1), we perform the overlap analysis of IRBL tools and investigate whether they show some complementary relationships with regards to being successful for specific sets of bug reports. Potential outcomes of this study are (1) all the tools are successful only for a specific set of bug reports so that we cannot identify any factor affecting the performance of each tool or (2) different tools are successful for each different set of bug reports so that we can identify what properties affect the performance of each tool.

5.3.5.1 Design

In this study, we consider the execution results provided in the reproduction study of Lee et al. [120] on Bench4BL. For each tool listed in Section 5.1, we collect its results (i.e., ordered lists of files suspected as bug locations) for every bug report of the projects listed in our dataset (cf. Table 6.1).

We then identify which tool successfully recommends files to fix for each bug report shown in Table 6.1. Since the results of IRBL tools are ordered lists of files, we take Top1, Top5, and Top10 recommended files when computing the performance of each tool. In this study, we classify whether an IRBL tool is successful for a specific bug report using Equation 5.7. Figure 5.2 presented with Venn diagrams [67] illustrates the relationships among state-of-the-art tools with respect to the number of bug reports that they can be localised exclusively or not at different positions (Top1, Top5, and Top10). For example, when we consider the diagram for Top1, BLIA makes correct localisation for 1804 bug reports, among which 160 are localised correctly only by BLIA at Top1 position.

The intersection among all tools tends to be larger when we increase N (of Top$N$). This indicates that recommendations by different IRBL tools gradually converge to a consensus as N is larger. However, increasing N inevitably results in more false positives and imprecise recommendations.

Although several bugs are correctly localised simultaneously by several tools, there is still a notable portion of the bugs that are localised exclusively by each of the tools. 13.0% (524 = 6+206+1+160+98+53), 6.6% (262 = 8+117+0+66+51+20), and 4.6% (174 = 6+89+0+37+27+15) of bug reports are successfully localised by only a single tool at Top1, Top5, and Top10 respectively. This implies that some tools appear to be exclusively successful for a specific subset of bug reports. If we can figure out the properties of the bug reports/source code files that make a specific tool perform better, we can tune the IRBL to fit the localisation decision depending on the bug report/source code file pair that is being assessed.

We perform an overlap analysis [174] between the tools for the bug reports that are localised correctly. The analysis is based on the following Equation 5.8:

$$\begin{aligned} C_{A\cap B} &= \frac{|C_A \cap C_B|}{|C_A \cup C_B|}\% \\ C_{A\setminus B} &= \frac{|C_A \setminus C_B|}{|C_A \cup C_B|}\% \\ C_{B\setminus A} &= \frac{|C_B \setminus C_A|}{|C_B \cup C_A|}\% \end{aligned} \tag{5.8}$$

where C_A represents the bug reports that have been correctly localised by tool A. $C_{A\cap B}$ is the percentage of the overlap between the sets of bug reports that have been correctly localised, while $C_{A\setminus B}$ is the percentage of the bug reports whose localisation is correctly found by A but missed by B.

5.3.5.2 Results

Table 5.3 shows the overlap analysis. The results quantitatively confirm the visual conclusions from the Venn diagrams of Figure 5.2.

Table 5.3: Overlap analysis results.

Tool		Top1			Top5			Top10		
A	B	$C_{A\cap B}$	$C_{A\setminus B}$	$C_{B\setminus A}$	$C_{A\cap B}$	$C_{A\setminus B}$	$C_{B\setminus A}$	$C_{A\cap B}$	$C_{A\setminus B}$	$C_{B\setminus A}$
BugLocator	Bluir	49.8	30.9	19.3	73.5	15.8	10.7	80.0	11.9	8.0
BugLocator	Amalgam	49.9	30.5	19.6	73.5	15.5	11.0	80.1	11.6	8.3
BugLocator	Brtracer	74.4	7.8	17.8	86.9	3.6	9.5	90.5	2.6	6.9
BugLocator	Blia	53.7	24.3	22.0	78.1	10.2	11.7	84.0	7.5	8.5
BugLocator	Locus	46.6	30.5	22.9	64.2	23.7	12.2	68.6	22.3	9.1
Bluir	Amalgam	98.8	0.1	1.1	99.3	0.0	0.6	99.4	0.0	0.6
Bluir	Brtracer	48.5	15.8	35.7	72.8	8.3	18.8	79.8	6.1	14.1
Bluir	Blia	44.7	23.1	32.2	72.7	10.4	16.9	79.9	7.6	12.5
Bluir	Locus	40.1	28.2	31.7	60.2	23.2	16.6	66.2	21.7	12.2
Amalgam	Brtracer	48.7	16.1	35.3	72.8	8.6	18.6	79.8	6.4	13.8
Amalgam	Blia	44.8	23.4	31.8	72.7	10.7	16.6	80.0	7.9	12.2
Amalgam	Locus	40.2	28.5	31.3	60.1	23.5	16.4	66.2	21.9	11.9
Brtracer	Blia	57.4	26.9	15.6	80.7	11.7	7.5	86.8	8.2	5.0
Brtracer	Locus	49.3	33.5	17.2	65.1	25.8	9.1	69.4	23.9	6.7
Blia	Locus	42.9	31.2	25.9	63.1	24.8	12.1	67.8	23.2	9.0

> *State-of-the-art tools tend to converge on suggesting the same locations for many bug reports when the list of recommendations is extended. Detailed comparisons among tools also reveal that some bug reports sets appear to be more localisable by specific tools.*

5.3.6 RQ-2: Feature Importance

Feature engineering is an important process for IRBL: tools in the literature mainly differentiate from each other on the variety of features that are considered. Indeed, the overall assumption is that the IRBL query formulation (*i.e., what information tokens from bug reports are used to search for relevant buggy files by matching appropriate information tokens from source code*) is one of the essential steps in bug localisation. Nevertheless, there is scarce knowledge about what are the most impactful query formulation schemes. To answer the second research question (RQ2), we investigate which information tokens are useful for an efficient query formulation. The objective is thus to assess the contribution of different combinations of tokens, across bug reports and source code files, in the success of bug localisation.

To answer our second research question (RQ2), we investigate first the MAP and MRR performance that can be achieved for each pairwise combination of features from bug reports and source code files to highlight the contribution of the features to MAP and MRR performance. Additionally, we perform an a-posteriori analysis for investigating the discriminating feature combinations that are effective for bug reports which are exclusively localised by specific IRBL tools. The objective of the a-posteriori analysis is to clarify our intuition that IRBL tools are most successful on specific sets of bug reports, which are correlated with the features used for similarity computation.

5.3.6.1 Design

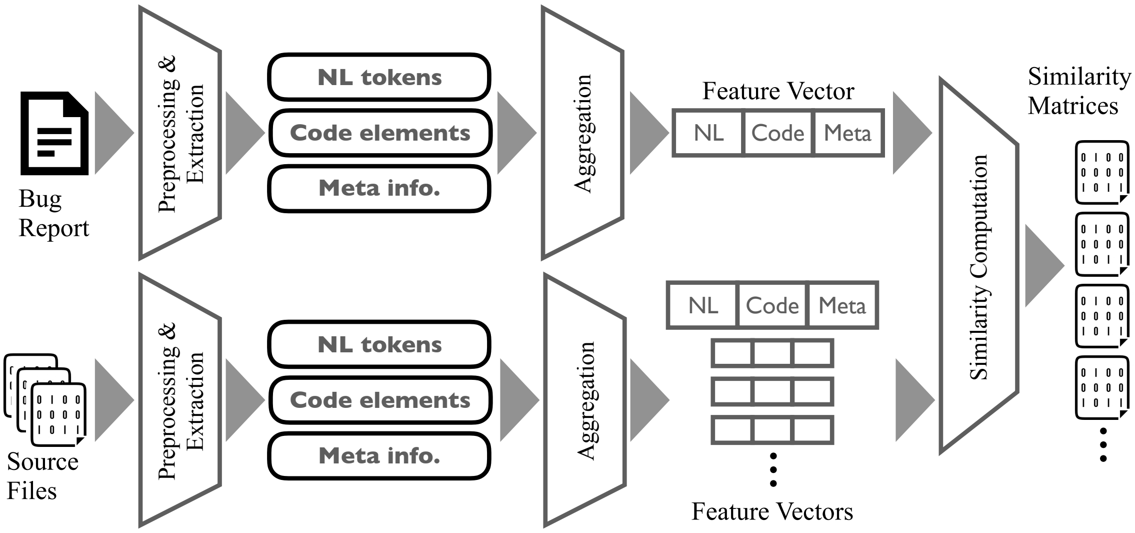


Figure 5.3: Feature engineering process.

To avoid biases of heuristics that various state-of-the-art tools have developed to boost performance in localisation, we implement a generic approach based on tasks and strategies that are commonly shared by different tools. This generic approach is illustrated in Figure 5.3. Bug reports and source code files of a project are processed to extract different types of tokens depending on the approach. These include natural language tokens (e.g., bug report description text and source code comments), code elements (e.g., method names in stack traces and source code AST attributes), and other metadata information (e.g., bug report submitter and source code file size). These information tokens are then used to build feature vectors for computing the similarity whose scores are leveraged to produce a ranked list of files that will be recommended for localising a given reported bug.

5.3.6.1.1 Preprocessing and extraction

As previously explained in Section 5.2, for our experiments, we have surveyed the features used in the literature and proposed to further refine some of them to more accurately study the impact of different information tokens. Overall, we consider 10 types of features from source code files (cf. Table 5.5) and 7 types of features from bug reports (cf. Table 5.4 along with the justification of use in bug localisation).

After identifying relevant tokens from bug reports and source code files, the tokenizer proceeds with a lexical analysis following the common steps from the literature, and which has influence in the ultimate performance of the retrieval model [199]: first, the text is retrieved, and tokens are produced, then stopword removal[5] is performed to reduce noise along with programming and project keywords. Finally, stemming (i.e., PorterStemmer [82]) is applied to all tokens to create homogeneity with the term's root (by conflating variants of the same term).

Given that terms from documents can be stack trace terms, code elements or natural language terms, we adopt slight specialisations in the preprocessing steps for the different types:

- Natural language: Tokenisation is based on white spaces as a classical separator. Tokens are then checked against the WordNet [160] dictionary to discard all unknown tokens.

[5] Stop words from NTLK framework: `https://www.nltk.org/`

Table 5.4: IR features collected from bug reports.

Feature	Description	Use in bug localisation
summary	The summary/title part of the bug report	Usually includes essential keywords about the problem.
description	The description part of the bug report	Is generally more verbose and provides additional descriptive tokens
rawBugReport	The whole bug report	Contains all textual information
stackTraces	The stack traces in the bug report	Entities (classes, files, etc.) in stack traces are likely buggy
codeElements	Code snippets in the bug reports	Can be indicative of the code part that is involved
summaryHints	localisation hints in the summary	Code-related terms in summary (e.g., package name) is relevant to buggy part
descriptionHints	localisation hints in the description	Code-related terms found by parsing description text (e.g., based on camel-Case regexp)

Table 5.5: IR features collected from source code files.

Feature	Description
packageNames	The parsed package names of the source code files
className	The parsed class names of the source code files
methodNames	The parsed method names of the source code files
methodInvocation	The parsed method invocation of the source code files
formalParameter	The parsed formal parameters of the source code files
memberReference	The parsed member references of the source code files
documentation	The parsed class names of the source code files
rawSource	Source file as a text
hunks	The hunks from the commits on the file
commitLogs	The commit logs of the file

- Stack traces and code elements: We use regular expressions to detect stack traces and code elements. Due to the specific nature of stack traces and code elements, the tokenisation is based on punctuations, camel case splitting (e.g., findNumber splits into find, number) as well as snake case splitting (e.g. *find_number* splits into "*find*, *number*").

5.3.6.1.2 Vectorisation and Similarity Computation

IRBL techniques recurrently treat source code as a form of text on which Natural Language Processing (NLP) techniques can be applied to extract features automatically. Several tools in the literature build upon the revised Vector Space Model (rVSM) [276] which represents bug reports and source code files as collections of tokens: these documents are then associated in a vector space model where tokens are weighted with term frequencies to calculate the similarity between documents.

In IRBL, we consider all project source code files as constituting the collection of documents d forming the search space, and a given bug report as the query q. tf-idf(t, d) assigns to term t a weight in document d that is

$$\text{tf-idf}(t, d) = \text{tf}(t, d) \times \text{idf}(t), \tag{5.9}$$

The document space is then defined as $D = \{d_1, d_2, \ldots, d_n\}$ where n is the number of documents in the corpus, t represents the terms, and tf (term frequency) is defined as follows:

$$\mathrm{tf}(t, d) = 1 + \log\left(f_{td} \right) \tag{5.10}$$

where f_{td} refers to the number of occurrences of a term t in document d. The idf (inverse document frequency) is defined by:

$$\mathrm{idf}(t) = \log\left(\frac{1+n}{1+|\{d : t \in d\}|}\right) + 1 \tag{5.11}$$

where $|\{d : t \in d\}|$ is the number of documents where the term t appears, when the term-frequency function satisfies $\mathrm{tf}(t, d) \neq 0$. The constant 1 is added to numerator and denominator of the idf, as if an extra document was seen containing every term in the collection exactly once, prevents divisions by zero.

Eventually, each document (bug report or source code file) is represented as a vector where each element corresponds to a term in the dictionary (of all terms appearing in the document space), together with a weight given by the tf-idf formula (Equation 5.9). Conservatively, the weight 0 is given to all dictionary terms that do not occur in a document. Ultimately, the vector form is used to calculate the relevance score between a document d (i.e., a source code file) and a query q (i.e., a bug report). This score is computed as the cosine similarity between the associated vector representations:

$$\mathrm{Similarity}(q_n, d_m) = \cos(q_n, d_m) = \frac{\vec{V}(q_n) \cdot \vec{V}(d_m)}{|\vec{V}(q_n)||\vec{V}(d_m)|}, \tag{5.12}$$

with $\vec{V}(q_n)$ being the vector of term weights of bug report n and $\vec{V}(d_m)$ being the vector of term weights of source code file m, where the term weights are computed using Equation 5.9.

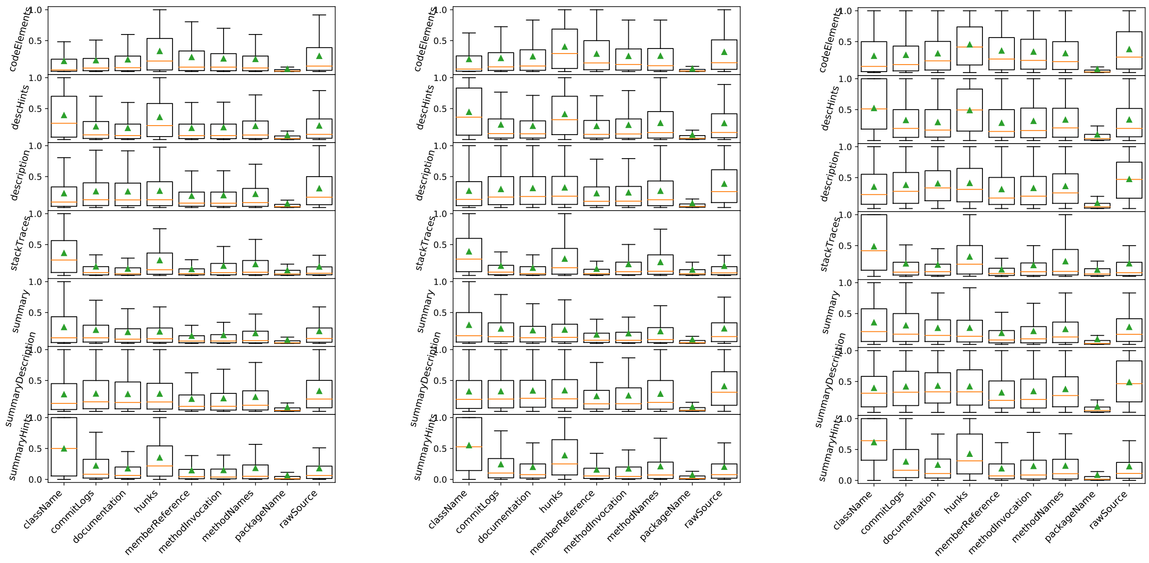
Bug localisation is then performed by assessing the strength of the similarity in a query-document pair (q, d)(cf. Equation 5.12). Concretely, we compute similarity matrices for each bug report and all source code files, by considering the pairwise combinations of features (cf. Tables 5.4 and 5.5) that are used to represent queries and documents.

5.3.6.2 Results

We investigate the distribution of MAP and MRR across various bug report sets, as illustrated in Figures 5.4 and 5.5. Given a bug report, we produce the ranked list of files to be recommended based on the similarity matrices: the higher the similarity, the higher the localisation rank. Our experiments are performed for different sets of bug reports that we regroup based on the performance of state-of-the-art tools to localise the reported bugs within project source code files. We consider the following three examples sets for building the insights for our subsequent approach:

- the set of bug reports that are *eventually localisable*, i.e., at least one state-of-the-art tool can place the correct localisation file as its Top1 recommendation.
- the set of bug reports which appear to be *suitable for IR-based bug localisation regardless of the tool.* These are then obtained by considering all bug reports for which all state-of-the-art Top1 recommended file was correct.
- the set of bug reports which are *only localisable by a single tool.* In this case, we focused on bug reports for which only BugLocator can provide a correct Top1 recommended source code file.

We note from the distributions of MAP and MRR values in Figures 5.4 and 5.5 that the range of distributions varies across the pairwise combinations of bug report/source code features. However, in most cases, the median values remain below 0.10 for the MAP as well as MRR. Nevertheless, it is noteworthy that a few combinations stand out by providing significantly high performance. This dissection

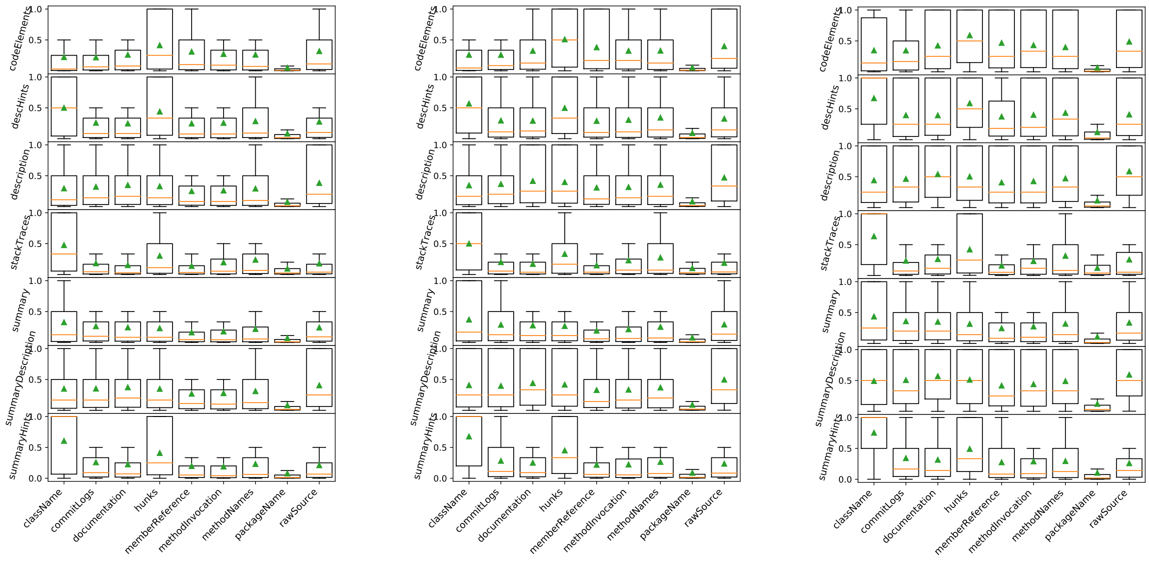


All bug reports for which _at least one_ state-of-the-art tool can recommend the correct bug location at Top1 of its list.

The set of bug reports which are only correctly localised _only by BugLocator_, and the location is recommended at Top1 of its list.

All bug reports for which _all_ state-of-the-art tools can recommend the correct bug location at Top1 of its list.

Figure 5.4: Mean Average Precision distributions for different sets of bug reports and with various query formulations (i.e., different combinations of features where the vertical axis refers to the bug report features and the horizontal axis to the source code features. The orange lines show the median values and the green arrows the mean values of the distributions).



All bug reports for which _at least one_ state-of-the-art tool can recommend the correct bug location at Top1 of its list.

The set of bug reports which are only correctly localised _only by BugLocator_, and the location is recommended at Top1 of its list.

All bug reports for which _all_ state-of-the-art tools can recommend the correct bug location at Top1 of its list.

Figure 5.5: Mean Reciprocal Rank distributions for different sets of bug reports and with various query formulations (i.e., different combinations of features).

illustrates how matching class names, hunks, raw code tokens from source code with code related tokens of bug reports are most effective for bug localisation. Indeed, `summaryHints/classNames`, `codeElements/hunks`, `summaryDescription/rawSource` appear to be the most successful pairs in terms of MAP and MRR.

> *Information tokens available in bug reports and source code files contribute with varying significance to the performance of bug localisation. Only a limited number of pairwise combinations among IR features yield relevant similarity scores for bug localisation.*

Finally, we observe that the different groups of bug reports that have been successfully localised by all or at least one or only one state-of-the-art tool show radically different MAP and MRR distributions for the various pairwise comparisons of features. This finding suggests that the considered sets of bug reports and/or the source code localisation files have distinctive attributes which make specific information tokens more or less important for similarity computation.

> *Although only a few IR features appear to be effective, it is noteworthy that IR feature pairwise combinations (between bug reports and code files) are not equally significant, in terms of contribution to the localisation performance, across the dataset.*

Figures 5.4 and 5.5 included detailed MAP and MRR for the dataset of bug reports localised exclusively by BugLocator. We have further performed an extensive a-posteriori analysis of feature importance for different sets bug reports that are exclusively localised by BugLocator, Bluir, Amalgam, BrTracer, Blia and Locus. We perform a Principal Component Analysis in which we rely on a LightGBM model to automatically compute feature importance as a classifier is trained to fit with the bug reports in specific sets. For each set of bug reports exclusively localised by a given IRBL tool, the feature importance values are averaged. Table 5.6 provides details on the number of principal components (features) which capture most of the variance (normalised importance), and the main contributors of those principal components (cumulative importance).

Importance provides a score that indicates how useful or valuable each feature was in the construction of the boosted decision trees within the model. The more an attribute is used to make key decisions with decision trees, the higher its relative importance.

This importance is calculated explicitly for each attribute in the dataset, allowing attributes to be ranked and compared to each other. It is calculated for a single decision tree by the amount that each attribute split point improves the performance measure, weighted by the number of observations the node is responsible for.

The feature importance is then averaged across all of the decision trees within the model as normalised importance. From the results, it appears that each exclusively-localised set has a different set of top important feature combinations. We can then conclude that these sets of bug reports require specific weighting scheme by the IRBL tool for the used features.

5.3.6.3 Implications

Findings from the dissection of the impact of IR feature selection on the performance of IRBL suggest two key implications for research:

1. **Similarity scores should be weighted**. Since IR features (and their pairwise combinations) have varying significance, the similarity scores that are computed should be weighted accordingly (*i.e., explicitly over-rank a bug report/code file pair when a significant feature combination has a relatively high score, and to avoid under-ranking a bug report/code file pair when an otherwise irrelevant feature combination has a low score.*). In the literature of IRBL, a number of tools (see reproducibility study in [120]) leverage a sampled dataset to compute the scores that guarantee

Table 5.6: Results of Principal Component Analysis.

	Feature	Importance	Normalised importance	Cumulative importance
Buglocator	summary2commitLogs	11.000	0.061	0.061
	summary2hunks	9.000	0.050	0.111
	summaryDescription2className	8.000	0.044	0.156
Bluir	summary2documentation	4.000	0.400	0.400
	descHints2className	3.000	0.300	0.700
	descHints2commitLogs	1.000	0.100	0.800
Amalgam	summary2className	11.000	0.193	0.193
	codeElements2className	9.000	0.158	0.351
	descHints2className	5.000	0.088	0.439
Brtracer	summary2commitLogs	33.000	0.058	0.058
	descHints2hunks	33.000	0.058	0.116
	summary2memberReference	32.000	0.056	0.172
Blia	summary2documentation	54.000	0.049	0.049
	summaryDescription2commitLogs	54.000	0.049	0.097
	summary2commitLogs	54.000	0.049	0.146
Locus	summaryDescription2commitLogs	124.000	0.077	0.077
	summaryDescription2methodInvocation	97.000	0.060	0.136
	descHints2className	91.000	0.056	0.193

the best localisation performance. These weighted scores are then used as generic scores for the overall approach.

2. **Weights of IR features should be adaptively computed for every specific set of bug reports**. Our experiments showed that relying on the similarity score for a pair of IR features (e.g., `summaryHints` in the bug report and `methodNames` in source code file) can lead to the varying performance of IRBL depending on the dataset. It is thus important to weigh these scores differently in accordance with the "nature" of the bug report and source code file that is being compared.

5.4 D&C: an Approach to Adaptively Learn the Weights of Similarity Scores

The implications enumerated in the previous section provide motivation for building a learning approach for adaptively computing the most effective weights to apply to the similarity scores of IR features of a given pair of bug report/source code file. We explore such an approach with a supervised learning technique where classification is built by learning from the examples of the dataset. However, building on the insights of our findings, we consider a multi-classifier approach where several classifiers are built, each being trained on specific parts of the dataset. During classification, instead of selecting a single classifier and using its probability outputs to present a ranked list of localised files, we combine the outputs of all classifiers by averaging the prediction probabilities.

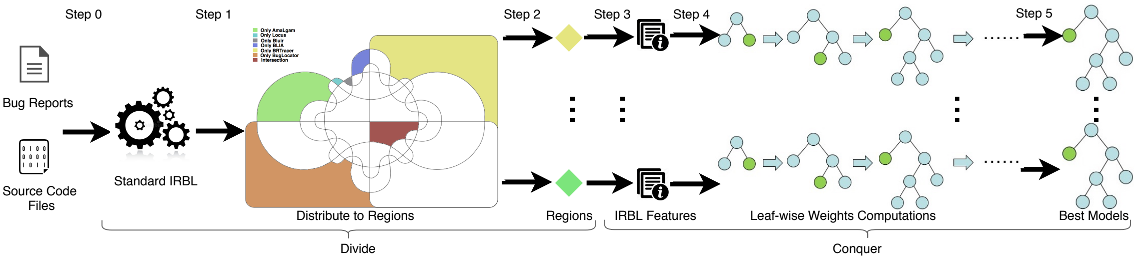


Figure 5.6: Divide and Conquer. Learning Approach.

5.4.1 Feature Space for the Classification Models

A classification model in machine learning-supported IR-based bug localisation is trained on a sample dataset to accurately suggest whether a given source code file is a likely location of the reported bug. Generally, this decision is taken by learning the relevant weights for the similarity scores and by computing the appropriate cutoff similarity threshold. In our case, we ignore the learned cutoff and return a ranked list of files based on the probabilities. In practice, we feed the learning algorithm with feature vector representations of the bug reports and source code files. We directly leverage the similarity matrices presented earlier, leading to 70-dimension feature vectors (with similarity scores of the pairwise combinations between the 7 bug reports information types and the 10 source code information types described in Section 5.3.6).

5.4.2 Divide-and-Conquer via Multi-classification

The first challenge for building a multi-classifier is to identify an effective way of splitting the dataset into *meaningful subsets*, i.e., subsets where samples share commonalities with respect to the significance of specific IR features for IRBL. To that end, an immediate approach would be to build metrics which first cluster the samples according to the distribution of MAP and MRR (based on a baseline IRBL tool) for pairs that provide similar localisation performance for the same feature subsets. This would require however an exhaustive exploration of combinations beyond the 70 pairwise combinations investigated in this study, as one would then need to investigate various combinations for $K = X + Y$ features, where $X \in [1..7]$ represent the number of bug report features and $Y \in [1..10]$ represent source code features.

Nevertheless, our empirical study of IRBL tools suggests that each tool appears to be more successful than others in some specific regions of the datasets while another large region seems to be localisable similarly by all tools. Given that our thorough review of the recent literature revealed that most state-of-the-art mainly differ by the set of features that are explored for the similarity computation, it would be reasonable to assume that the regions represent sets of bug reports/code files where the computed weights are effective. Thus, we propose to directly leverage the assessment result of state-of-the-art tools as a metric to split the training dataset.

Figure 5.6 overviews the overall divide-and-conquer approach. We start by running and assessing state-of-the-art tools on Bench4BL, and delimitating different regions of the datasets according to the performance of each tool in comparison with others. Then, for each of such regions, we apply the learning process formally detailed in Algorithm 3. The regions contain the list of bug reports ids, thus we start by extracting the fine-grained 7x10 IRBL features (f), and their corresponding binary labels (l) for the bug report/code files pairs. In a given project, only a few source code files are buggy ($trueLabels$) and most of them are not ($falseLabels$) which is creating a highly imbalanced dataset. To handle this imbalance nature, we calculate a frequency coefficient ($freqCoef$) that represents the ratio of the false labels observed in the training data for each project. Then, we apply the frequency coefficient over the labels observed in the training data to compute the class weights (cw). In the class weight computation, we add 1 to the label values, in order to prevent having the class weights of falseLabels all zero. We encode the features, labels and class weights into dataset object (*dtrain*). We retrieve hyperparameters of the training (*p*) that are previously calculated using a grid search approach. We train the region classifier for 10000 iterations, with early stopping of 10 rounds until a best model is found. Finally, we predict the testing data on the best model and save the prediction probabilities.

We consider specific subsets of the dataset for training since they are presenting the following properties:

1. bug reports that fit a specific tool (e.g., *Brtracer* represents the subset of bug reports which were accurately localised by Brtracer).
2. bug reports are exclusively fit to a specific tool (e.g., $Only - Brtracer$ represents the subset of bug reports which were accurately localised by only Brtracer).
3. bug reports that appear to be rich in terms of information and are thus easy to localise by any tool ($INTER$).
4. bug reports that are localisable at the Top1 position by at least one tool ($UNION$).
5. bug reports that do not seem to contain enough information to be accurately localised by any tool ($\neg UNION$).

Algorithm 3: Divide and Conquer Learning Algorithm

```
input  : A region $r$ from regions $R$ which is a list of bug reports, where $r \subset R$ and $R \leftarrow \{$
         Only-Brtracer,Only-LOCUS, Only-Blia,Only-Amalgam, Only-Bluir,Only-BugLocator,
         Intersection, Top-1s,Non-Top1s $\}$
input  : A region $\neg r$, where $\neg r = R \setminus r$
output : Predicted probabilities
for $r$ to $R$ do
    $trains,tests \leftarrow$ timelineSplit($r$);
    for $train,test$ in $trains,tests$ do
        $f,l \leftarrow$ extractFeaturesAndLabels($train$);
        $cw \leftarrow$ calculateClassWeights($l$);
        $dtrain \leftarrow$ createDatasetObject($f,l,cw$);
        $p \leftarrow$ getHyperparameters();
        $best\_model \leftarrow$ trainModel($dtrain,p$);
        $prediction\_probabilities \leftarrow$ best_model.predict($test$);
        return $prediction\_probabilities$
Function calculateClassWeights (l: labels) : classWeights
    $trueLabels \leftarrow$ filter($l$,1);
    $falseLabels \leftarrow$ filter($l$,0);
    $freqCoef \leftarrow$ count($falseLabels$) / count($trueLabels$);
    $classWeights \leftarrow (1 * freqCoef) + 1$ ;
    return $classWeights$;
```

Splitting the dataset into test and training. For each selected region r, we first prepare the dataset for validation by splitting it into validation and training data based on the bug report creation year (i.e., *timelineSplit* in Algorithm 3): we train on bug reports with creation date less than X and validate on creation date greater than X, where X $\in \{1-1-2008, 1-1-2009, ..., 1-1-2016\}$.

Handling dataset imbalance. The whole dataset is highly imbalanced since only a very small portion ($\approx$ 20 000 out of 35 088 421) of the bug report-source code pairs are actually buggy (*i.e. minority class*), whereas the majority of the pairs are non-buggy (*i.e. majority class*). Machine learning classifiers have a bias towards classes which have a large number of instances and thus tend only to predict the majority class data. The features of the minority class are then treated as noise and are often ignored. Therefore, there is a high probability of misclassification of the minority class as compared to the majority class, and hence the likelihood of poor classification for bug localisation. Dealing with imbalanced datasets entails strategies such as either improving classification algorithms, or balancing classes in the training data before providing the data as input to the machine learning algorithm. We initially experiment on balancing the classes in the training dataset by applying resampling techniques (Random Under-Sampling, Random Over-Sampling, Cluster-Based Over Sampling [264], SMOTE [34], MSMOTE [71]). The main objective of these techniques is to *artificially* balance classes by either increasing the frequency of the minority class or decreasing the frequency of the majority class. However, we note that resampling techniques are not very effective since these random resampling techniques are *overfitting* the training data, and others (Cluster-Based Over Sampling [264], SMOTE [34], MSMOTE [71]) are not performing well due to the high dimensionality of data (70-dimension feature vectors).

We propose to address the imbalance dataset issue by adapting the classification algorithm. Our machine learning classification is performed with the LightGBM gradient boosting framework [83]. LightGBM documentation[6] provides parameters to explicitly instruct the algorithm to account for data imbalance. It is further known in the practice of machine learning (e.g., Kaggle competitions) to be effective for dealing with imbalanced dataset[7]. LightGBM supports weighted training, which uses the observation weights of classes for bias correction. We compute the class weights as described in Algorithm 3. We filter the training data by selecting positive and negative samples where the positive samples have the label 1, which presents the actually buggy bug report-source code pairs (*i.e., minority class*), and negative samples have the label 0, that presents the non-buggy pairs (*majority class*). We calculate the frequency coefficient, which is a parameter that is inversely proportional to class frequencies in the training data for each project. Finally, we calculate the class weights by applying the frequency coefficient over the labels observed in the training data. During training, LightGBM, which uses a Leaf-wise (Best-first) Tree Growth approach will choose the leaf with max delta loss to grow. Due to the larger loss function pre-factor (*i.e., class weights*), the classes with higher weights matter more, even they are minority class.

Hyperparameter optimisation. LightGBM uses the leaf-wise tree growth algorithm, while many other popular tools use depth-wise tree growth. Compared with depth-wise growth, the leaf-wise algorithm can converge much faster. However, the leaf-wise growth may be over-fitting if not used with the appropriate parameters. To build optimal models using a leaf-wise tree growth approach, there are a few important parameters to tunes:

- Number of estimators. The number of trees in the model.
- Number of leaves. This is the main parameter to control the complexity of the tree model. Theoretically, we can set $num\_leaves = 2^{(max\_depth)}$ to obtain the same number of leaves as a depth-wise tree. However, this simple conversion is not good in practice. The reason is that a leaf-wise tree is typically much deeper than a depth-wise tree for a fixed number of leaves. Unconstrained depth can induce over-fitting. Thus, when trying to tune the *num_leaves* parameter, we should let it be smaller than $2^{(max\_depth)}$.
- Learning rate. This a hyper-parameter that controls how much we are adjusting the weights of our network with respect to the loss gradient.
- Feature fraction. The fraction of observations to be selected for each tree. Selection is made by random sampling.

We used a grid search which systematically works through multiple combinations of parameter tunes, cross-validating all runs to determine which one gives the best performance. Eventually, for the training, a binary classifier is constructed with $learning\_rate = .03$, $n\_estimators = 100$, $num\_leaves = 31$ and $feature\_fraction = .08$.

For runtime efficiency, we have transformed our training data into LightGBMs' inbuilt dataset object where the features, labels, and weights are represented in a memory-efficient binary form.

[6] https://lightgbm.readthedocs.io/en/latest/Parameters.html

[7] https://www.kaggle.com/pranav84/lightgbm-fixing-unbalanced-data-lb-0-9680

Models Selection and Prediction. We train each classifier in 10 000 iterations, with *early stopping*, which is a form of regularisation used to avoid overfitting when training a learner with an iterative method. Early stopping works by monitoring the performance of the model that is being trained on a separate test dataset and stopping the training procedure once the performance of the test dataset has not improved after a fixed number of training iterations. It avoids overfitting by attempting to automatically select the inflexion point where performance on the test dataset starts to decrease while performance on the training dataset continues to improve as the model starts to overfit. Concretely, we can check whether there is no improvement via the root mean squared error (RMSE) over the 10 consecutive iterations during the training. Once there is no improvement, the training early stops since the *best_model* is found (or in the worst-case scenario the iteration count reaches to 10 000 which terminates the training by selecting the last iteration as *best_model*).

Once the *best_model* for each classification training is found, we apply it to the relevant test dataset, yielding probability values for all pairwise combinations of bug report-source code files. A high probability value implies that the classifier highly recommends the source code file in a pair to be relevant to the associated bug report.

5.4.3 Ranking of Bug Localisation Recommendations

The prediction probabilities of the source code files in each model are combined into a single ranking by averaging the prediction probabilities. We use a classical incremental ranking, ties are ignored and the ranks are assigned in the order they appear. For example, when files A and B have the same prediction probability, the classical incremental ranking ranks A in rank r, and B in rank $r + 1$.

5.5 Assessment

The assessment of this work investigates execution times and performance results in validation experiments. We also perform a comparison against the state-of-the-art and study the impact of the choice of multi-classification.

5.5.1 Execution Times

The training of 140 classifiers and all ($\sim$ 70 000) predictions took 69,25 hours on a server with 110 Intel Xeon E5-4650v2@2.4GHz with 4TB of RAM.

The feature extraction processes are executed separately in parallel for every project. The execution speed is roughly proportional to the number of source code files in the project and the number of bug reports. Most of the small projects (projects having less than a million bug report-source code pairs) finish within an hour, while the bigger projects JBOSS-WFLY, APACHE-HIVE took around 50 hours to complete the feature extraction. For the biggest project APACHE-CAMEL, we executed the feature extraction process (including parsing) more than 80 hours on our 96 Intel Xeon E5-4650v2@2.4GHz with 4TB of ram.

5.5.2 Validation experiment

To assess the `D&C` approach, we propose a validation experiment where the dataset is split around per year. Bug reports (and associated localisation information) created before 1 January of the selected year are considered as training data. All bug reports from 1 January and onwards are used for testing (i.e., validation) of the model. Table 5.7 shows the validation experiment results for each year. Overall, the training dataset includes 5321 bug reports, and the built localiser is tested on

3954 bug reports. We record an MAP of 0.507 and an MRR of 0.617. It should be noted that this performance is obtained with a cleaned dataset (i.e., where considered bug reports for validation are not post-fixing activities).

Table 5.7: Validation Experiment.

	# bug reports	2008	2009	2010	2011	2012	2013	2014	2015	2016	Average
	in training	79	368	825	1293	1619	1936	2328	3102	4213	Perf.
	in validation	220	345	366	262	251	316	565	834	795	(mean)
	MAP	0.504	0.443	0.511	0.589	0.553	0.562	0.495	0.494	0.494	**0.52**
	MRR	0.635	0.571	0.624	0.693	0.646	0.678	0.603	0.602	0.596	**0.63**
% localized	Top1	52%	43%	50%	56%	52%	57%	49%	48%	47%	**50%**
	Top5	75%	75%	77%	84%	79%	80%	74%	75%	74%	**77%**
	Top10	84%	85%	85%	90%	86%	87%	80%	83%	83%	**85%**

On average, we find that 50% of bugs can be localised at Top1 by `D&C`, and 77% of bugs at Top5 while 85% of bugs are detected at Top10.

5.5.3 Comparison against the state-of-the-art

To compare `D&C` against the state-of-the-art, we consider the execution results of IRBL tools compiled by Bench4BL and focus on results of bug reports that are part of our cleaned validation set (i.e. those bug reports that are truly pre-fix / fix-independent reports). Given that Bench4BL only provides information about bug reports in Top10 for each tool, our comparisons are based, for each tool, on a different set. Table 5.8 provides performance comparison details about MRR and MAP. We note that `D&C` provides a substantial performance improvement of MAP and MRR over all tools: MAP is improved by between 4 and up to 10 percentage points, while MRR is improved by between 1 and up to 12 percentage points. Furthermore, we note that `D&C` generally manages to localise more bug reports at Top1 (e.g., 20% more than Blia), Top5 (e.g., 15% more than Bluir) and Top10 (e.g., 13% more than Bluir).

Table 5.8: Performance comparison against state-of-the-art IRBL tools. Dataset are cleaned to fit our criteria on pre-fix activities.

		Performance		% Localised Bug Reports		
Tool	# Bug Reports	MAP	MRR	Top1	Top5	Top10
BugLocator	3949	0.425	0.546	42%	69%	78%
D&C	3949	0.507	0.617	50%	76%	84%
Brtracer	3949	0.461	0.599	49%	74%	81%
D&C	3949	0.507	0.617	50%	76%	84%
Bluir	3947	0.402	0.493	37%	64%	74%
D&C	3947	0.507	0.617	50%	76%	84%
Amalgam	3952	0.405	0.496	37%	65%	74%
D&C	3952	0.508	0.617	50%	76%	84%
Blia	3879	0.434	0.554	43%	71%	80%
D&C	3879	0.514	0.625	50%	77%	85%
Locus	3362	0.422	0.604	49%	75%	83%
D&C	3362	0.506	0.618	50%	76%	85%

Table 5.8 further details the detection performance of the tools in terms of percentage of localised bug reports at Top1, Top5 and Top10. D&C localises up to 13% more bugs at Top1 than the state-of-the-art tools, between 1% and up to 11% more bugs at Top5.

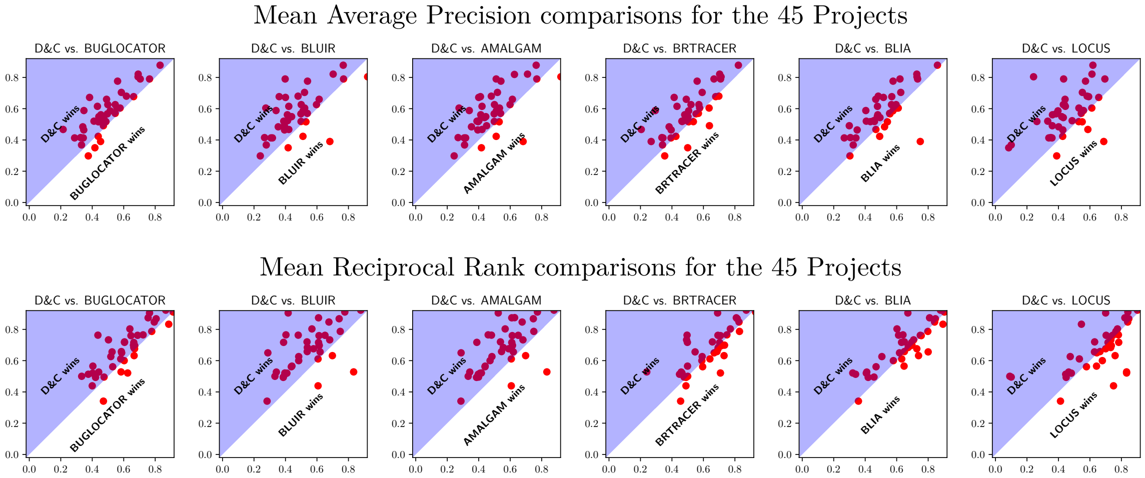


Figure 5.7: Project-wise performance comparison(X and Y axes show MAP and MRR values, the red dots).

5.5.4 Project-wise performance comparison

Given the lack of cleaned (i.e., pre-fix activity) data for many projects, our learning merges data from all bug reports for training. We now investigate the performance on D&C on localising bug reports for each project (with training on all project data).

Table 5.9 provides details on the performance obtained for each project in our dataset. It is noteworthy that MAP and MRR performance are significantly varying across projects. While MAP can drop to as low as 0.35 for some projects (e.g., APACHE-WEAVER), it can reach 0.8 for others (e.g., APACHE-LANG). APACHE-IO shows an MRR of 0.91 while JBOSS-JBMETA has an MRR at 0.34. The data further shows that this performance is not correlated with the size of the bug reports set of the project.

We compare the MAP and MRR values per project with that of the state-of-the-art IRBL tools for these projects. Figure 5.7 illustrates how D&C outperforms every other considered state-of-the-art approach for the large majority of projects.

5.5.5 Impact of multi-classification.

Our D&C approach builds and merges the results of multiple classifiers to rank localisation files. We investigate the performance of specific classifiers such as the ones built by considering only datasets where a given state-of-the-art tool is exclusively performing well (e.g., Only-Brtracer), or the $INTER$section of datasets where state-of-the-art tools are performing well, or datasets of $UNION$ of bug reports that are localised with Top1 predictions by any state-of-the-art tools, or datasets of $\neg UNION$ of bug reports which are localised in Top1 by none of the state-of-the-art tools. We refer to these classifiers as region-specific classifiers. Table 5.10 provides MAP and MRR results along with that of D&C. We observe that D&C outperforms every other region-specific classifier, followed by UNION. This result suggests that the D&C learning model finds a good way to compute the most effective weights to apply to the similarity scores of IR features. On the other hand, $INTER$ performs lower than $UNION$ perform very well due to the under-representation of a diverse set of

Table 5.9: Project wise performance results.

		Performance		% Localized Bug Reports		
Project	# Bug Reports	MAP	MRR	Top1	Top5	Top10
APACHE-CAMEL	797	0.434	0.569	44%	72%	80%
APACHE-CODEC	28	0.79 0	0.923	86%	100%	100%
APACHE-COLLECTIONS	26	0.776	0.874	81%	92%	96%
APACHE-COMPRESS	56	0.704	0.847	77%	95%	100%
APACHE-CONFIGURATION	11	0.518	0.611	45%	73%	82%
APACHE-CRYPTO	1	1	1	100%	100%	100%
APACHE-CSV	8	0.66 0	0.656	38%	100%	100%
APACHE-HBASE	228	0.520	0.626	52%	77%	84%
APACHE-HIVE	395	0.413	0.513	39%	65%	75%
APACHE-IO	43	0.878	0.911	84%	100%	100%
APACHE-LANG	99	0.821	0.868	84%	100%	100%
APACHE-MATH	4	0.819	0.875	75%	100%	100%
APACHE-WEAVER	1	0.350	0.500	0%	100%	100%
JBOSS-ELY	5	0.467	0.733	60%	80%	100%
JBOSS-ENTESB	6	0.423	0.439	17%	83%	83%
JBOSS-JBMETA	10	0.298	0.341	30%	30%	40%
JBOSS-SWARM	35	0.464	0.52 0	34%	77%	86%
JBOSS-WFARQ	1	1	1	100%	100%	100%
JBOSS-WFCORE	310	0.415	0.527	41%	67%	79%
JBOSS-WFLY	461	0.413	0.493	37%	62%	72%
JBOSS-WFMP	3	0.39 0	0.528	33%	100%	100%
SPRING-AMQP	67	0.561	0.720	58%	87%	93%
SPRING-ANDROID	7	0.588	0.717	58%	87%	93%
SPRING-BATCH	239	0.543	0.684	58%	87%	93%
SPRING-BATCHADM	15	0.484	0.565	33%	87%	93%
SPRING-DATACMNS	110	0.626	0.76 0	66%	89%	94%
SPRING-DATAGRAPH	2	1	1	100%	100%	100%
SPRING-DATAJPA	103	0.586	0.762	65%	92%	96%
SPRING-DATAMONGO	195	0.567	0.699	56%	87%	94%
SPRING-DATAREDIS	37	0.626	0.787	68%	92%	95%
SPRING-DATAREST	72	0.549	0.658	53%	82%	89%
SPRING-LDAP	29	0.516	0.632	45%	86%	97%
SPRING-MOBILE	11	0.804	0.833	73%	100%	100%
SPRING-ROO	113	0.517	0.561	42%	75%	84%
SPRING-SEC	242	0.604	0.676	55%	83%	94%
SPRING-SECOAUTH	14	0.604	0.65 0	57%	71%	79%
SPRING-SGF	19	0.615	0.716	58%	95%	95%
SPRING-SHDP	14	0.672	0.763	71%	86%	86%
SPRING-SOCIAL	35	0.677	0.765	66%	94%	94%
SPRING-SOCIALFB	11	0.68 0	0.803	73%	100%	100%
SPRING-SOCIALLI	4	0.491	0.521	25%	100%	100%
SPRING-SOCIALTW	7	0.79 0	0.905	86%	100%	100%
SPRING-SPR	25	0.368	0.495	36%	64%	72%
SPRING-SWF	40	0.523	0.600	50%	73%	78%
SPRING-SWS	66	0.57 0	0.696	59%	82%	88%

bug reports. Finally, we note that region-specific classifiers targeting datasets that are best fitted to a given state-of-the-art tool are not performing well as well. This again suggests that a significant portion of bug reports are not adapted to each such classifiers. Finally, we note that the classifier based on $\neg UNION$ (i.e., no state-of-the-art approach is successful for Top1) can lead to a better performance than some other region-specific classifiers. This finding again confirms that it is indeed necessary to triage the dataset (i.e., dividing based on some rationale).

Table 5.10: Region-classifiers vs Multi-classifier.

	Performance		% localised Bug Reports		
	MAP	MRR	Top1	Top5	Top10
Region-classifiers					
OnlyBugLocator	0.323	0.404	27%	56%	67%
OnlyBLUiR	0.008	0.009	0.1%	1%	1%
OnlyAmaLgam	0.207	0.261	18%	36%	44%
OnlyBRTracer	0.348	0.438	29%	61%	73%
OnlyBLIA	0.367	0.454	33%	61%	71%
OnlyLocus	0.421	0.517	39%	68%	77%
BugLocator	0.483	0.595	48%	74%	82%
BLUiR	0.477	0.589	47%	73%	81%
AmaLgam	0.476	0.588	47%	74%	81%
BLIA	0.480	0.593	48%	73%	81%
BRTracer	0.479	0.594	48%	73%	81%
Locus	0.469	0.579	46%	73%	81%
UNION	0.483	0.598	48%	74%	82%
¬UNION	0.352	0.445	31%	62%	73%
INTER	0.464	0.568	45%	71%	79%
Multi-classifier					
D&C	0.507	0.617	50%	76%	84%

5.6 Discussion

We discuss the insights of our study, the practicality of D&C and the threats to the validity of our results.

5.6.1 Insights

On the dividing strategy. Given the challenge in categorising bug reports with respect to localisation performance, we leveraged state-of-the-art prediction results as proxies to identify groups of bug reports which may share similar properties that are relevant to localisation. Concretely, we consider that bug reports, exclusively detected by a given tool (or detected by all tools), share some common characteristics which fit with the different feature sets used by different tools. A potential research direction would consist of further investigating the metrics that could be used to implement other dividing strategies.

On the considered features. In this work, we focus on common, easily extractable features used by most works in the literature. Nevertheless, we note that there are several recent works which propose other specific features such as code smells [218] or function call graphs [250]. Although these features have not led to significant improvement of localisation performance in one-size-fits-all approaches, they could have a more positive impact in the D&C approach since region-specific training could properly weight the similarity scores associated to such features for related corner-case bug reports.

5.6.2 Practicality

Validation experiments (see Table 5.7) provide evidence that the D&C approach is stable: whether training data is small (e.g., 79 bug reports in 2008) or huge (e.g., 4216 bug reports in 2016), the overall performance is stable. In practice, the training phase which is the most time-consuming can then be done once and be regularly applied to new bug reports. Because some projects have only one bug report, we have opted to merge all data in a cross-project training scenario. The yielded results are promising and show that D&C can be leveraged from the start of a development project since training data can be borrowed from other projects. Finally, note that we have performed in-project training as well for big projects such as APACHE-CAMEL: the obtained performance results are similar to when using cross-project training.

5.6.3 Threats to Validity

External validity. Our study carries a few threats to validity related to the use of Bench4BL where the ground truth of localisation may be incomplete (a given bug report may have been fixed by more commits than included in the bench), wrong (some localised files may be wrong as the commit could have been reverted later). The quality of the bug reports may also bias the experiments. Finally, we focus only on Java and the D&C approach may not generalise to other languages. Nevertheless, these threats are mitigated by the size of the benchmark as well as the inclusion of projects which have been largely investigated in other software mining research works.

Internal validity. Our work also carries a number of threats related to the process of cleaning the dataset to consider only post-fix activities. We minimise this threat by using heuristics that are reasonable given the practice of bug reporting in open source communities. Another threat to internal validity is the selection of LightGBM as the core supervised learning algorithm. There is a need to investigate in future work, whether other algorithms will lead to the same conclusions about dividing and conquering with multi-classification. Finally, the presented results are based on merging prediction probabilities without any form of normalisation. Actually, we have used some heuristics to normalize the predictions, but did not notice any change in the performance score.

Construct validity. In this study, we hypothesise that the weighting scores of features are the key elements for improving bug localisation. However, one threat to validity is that we have leveraged machine learning to estimate these weights: our training step may not actually be modelling the features weights. Finally, we are focusing on comparing the tools, with the threat to validity that the major issue could be rather in the inner IR method.

Conclusion validity. After dataset curation to remove post-fix activities, the available validation sets contain each a few hundreds or thousands bug reports. Our conclusions are thus threatened. Furthermore, our bug reports may not be heterogeneous indeed. Nevertheless, we minimised these threats by considering the largest dataset ever used in the literature of IRBL.

5.7 Related Work

5.7.1 IR methods

Information Retrieval (IR) generally implies a method used to find data related to a user's needs in various groups of text data. Some of the bug localisation studies also use the IR-based technique to find the source code that should be modified using the information from the bug report. Various approaches are proposed including both simple text models such as Unigram Model (UM) [212], Vector Space Model (VSM) [202, 248] and sophisticated models such as the Latent Semantic Analysis Model (LSA) [42, 46], the Latent Dirichlet Allocation Model (LDA) [25], and the Cluster Based

Document Model (CBDM) [256]. As Rao and Kak [193] revealed, simple techniques can provide higher accuracy than sophisticated techniques, leading to wide use of VSM techniques in recent works.

Thomas et al. [223] focused on the impact of classifier configurations, studying several parameter values (e.g., code pre-processing, similarity metrics, and term weights) in bug localisation task to investigate the performance of the underlying IR methods. Khatiwada et al. [87] investigated the performance of a new paradigm of information-theoretic IR methods(Pointwise Mutual Information (PMI) and Normalised Google Distance (NGD))in bug localisation tasks.

5.7.2 Query Reformulation

Sisman and Kak [208] introduced query reformulation in the context of IR-based bug localisation. Chaparro et al. [32] manually reduce noisy, ineffective queries to reformulated queries that contain only terms that describe observed behaviours, and find that the reformulated queries have much-improved performance. Rahman et al. [192] incorporate context-aware (i.e., report quality-aware) query reformulation into the IR-based bug localisation.

5.7.3 VSM in IRBL

Variations of Vector Space Models (VSMs) are used for bug localisation as well. BugLocator [276] uses the revised Vector Space Model (rVSM) to recommend target files to be fixed. They first made a vector by using keywords extracted from the incoming bug report and then compare the vectors to recommend the most probable source code. The technique computes *SimiScore*, a metric that calculates a similarity between an incoming report and the files fixed by previous bug reports. Wang et al. [234] combined a genetic algorithm and VSM to improve the performance of IRBL. They used Eclipse, SWT, AspectJ and ZXing projects to evaluate their approach. In the evaluation, this technique achieves 33–48% accuracy, outperforming previous bug localisation approaches.

5.7.4 Topic modelling in IRBL

Topic modelling and semantic analysis are common techniques used in IRBL. PROMESIR [188] utilises Latent Semantic Analysis (LSI) [42] to identify buggy files. Lukins et al. [143] adopted Latent Dirichlet Allocation (LDA) [25] to their approach that models source code topics and showed its effectiveness with a small number of case studies. BugScout [170], on the other hand, builds topic models for both source code and bug reports and compares their distribution to locate files to fix a bug. Takahashi et al. [218] use code smells to improve bug localisation.

5.7.5 Stack traces in IRBL

Stack traces are regarded as a promising information source in bug localisation. Wong et al. [247] proposed a Brtracer which further considers stack traces in similarity scores. Lobster [166] also uses stack traces to compare with code elements in source code files. CrashLocator [250] focuses more on stack traces together with function call graphs.

5.7.6 Feature combinations in IRBL

Combining existing approaches can improve the performance of IRBL. Amalgam [232] uses version history information building on the intuition that bugs are likely to occur again in files changed more frequently. BILA [267] takes advantage of source code entities [199], *SimiScore* [276], stack trace score [247], and version history score. However, it is difficult to compare whether they have improved on average since there is no experiment on the Eclipse project which is the biggest bug reporting system. Locus [242], the most recent technique, proposes fine-grained localisation by using commit logs as well as change hunks in revision history to improve similarity measures. Then, the technique incorporates token, code entity, and history scores to compute the final similarity score. The evaluation results suggest that the technique outperforms the existing techniques by 8–10% and achieves up to 64% accuracy. They used SWT, JDT, Tomcat projects to evaluate the approach and obtained around 8% to 10% point higher on average and maximum 64% accuracy.

5.7.7 New approaches to IRBL

Deep learning techniques also can be leveraged together with IR techniques for bug localisation. Lam et al. presented HyLoc [103] and DNNLoc [104]. These approaches use deep neural networks to learn relevancy between tokens in bug reports and code elements in the source code. In addition, the approaches add an autoencoder to reduce the size of the features. Since the number of tokens in bug reports and source code is often hundreds of thousands, the scalability of neural networks is limited. The autoencoder compresses dimensions of input features.

Other IRBL techniques consider machine learning. Ye et al. [263] proposed a learning-to-rank approach to bug localisation based features representing the degree of suspiciousness. Kim et al. [89] dealt with bug report quality to improve bug localisation with a two-phase model focusing on high-quality bug reports.

5.7.8 IRBL-related studies.

Closely related to our work, Le et al. [108, 109] have proposed a study where they attempt to predict whether the ranked list produced by a bug localisation tool is likely to be relevant to the given bug. They extract various textual and metadata features from 3 old projects and test on two IRBL techniques. They indeed find that it is possible, to some extent, to predict the effectiveness of the considered techniques. Our work is a generalised and large-scale investigation into the question of IRBL performance.

Saha et al. [198] conducted a study investigating the applicability of an IRBL technique on non-object-oriented code, notably C programs. They extend a previous approach targeting Java programs to support C code parsing. They found that IR-based bug localisation in C software at the file level is overall as effective as in Java software. They, however, conclude that using program structure information to tune localisation is less relevant to C software than for Java software.

Wang et al. [231] have conducted an analytical study and a user study on IRBL techniques to assess their usefulness. Focusing on a single technique, BugLocator, and four common projects from previous studies, they report that the information needed for IR-based techniques to be effective is often not available in bug reports. Their user study further suggests that even when high-quality bug reports are available and IR-based techniques can "perfectly" rank bug locations, they may still benefit developers only marginally since high-quality bug reports are often good enough to guide developers to the file, which can be located without any additional help. They also discuss that suspicious file ranking by IRBL techniques may not help speed up the localisation of the bug within that file, which could be the most time-consuming part of debugging. Nevertheless, as in many software engineering-related tasks, automated IRBL can accelerate the realisation of other endeavours (e.g., improve the scalability of automated program repair).

Recently, Lee et al. [120] have proposed an extensive benchmark for IRBL. They used this benchmark to offer a clear view of the performance of state-of-the-art working tools. In our study, we leveraged their dataset and further curated it to remove any post-fix activities data.

5.8 Summary

We have proposed D&C, a novel IRBL approach which adaptively learns to compute the weight to associate to similarity scores of IRBL features. To that end, we leverage a gradient boosting supervised learning technique to build multi-classifiers by training on homogeneous subsets of bug localisation datasets. In practice, we have performed a large scale empirical study which revealed that state-of-the-art tools, which mainly differ by the features that are considered, appear to be fit for specific bug reports. Thus, we leverage the assessment results of six state-of-the-art tools as a metric for splitting the dataset and allowing a meaningful training of specialised classifiers whose outputs are then combined to produce an accurate ranking of localisation recommendations. Comparing to state-of-the-art tools, D&C shows higher performance on Bench4BL, currently the most comprehensive bug localisation dataset in the literature. Typically, our validation experiments yield an MAP score of 0.52, and an MRR score of 0.63 with a curated version of Bench4BL. Comparison against the state-of-the-art shows that D&C provides a substantial performance improvement of MAP and MRR over all tools: MAP is improved by between 4 and up to 10 percentage points, while MRR is improved by between 1 and up to 12. Finally, we note that D&C is stable in its localisation performance: around 50% of bugs can be located at Top1, 77% at Top5 and 85% at Top10.

Future along this direction could consider including more classifiers trained on corner-case bug reports which can be discovered by tools which include features such as crashes or code smells. Similar experiments can be performed at the method level to assess the performance on finer-grained bug localisation as attempted by Locus, although with poor performance. Finally, the research community can benefit from reverse engineering of the exclusive successful localisation results by various state-of-the-art to formally model the characteristics of the associated bug reports, to improve other research lines, notably on duplicate bug detection, bug triaging, etc.

5.9 Bug Report driven Program Repair

5.9.1 Overview

Automated program repair (APR) has gained incredible momentum since the seminal work of GenProg [238], various approaches [35, 39, 72, 76, 84, 88, 110, 111, 117, 128, 131, 132, 137, 138, 140, 155, 172, 238, 241, 255, 257] have been proposed in the literature aiming at reducing manual debugging efforts through automatically generating patches. Beyond fixing *syntactic errors*, i.e., cases where the code violates some programming language specifications [63], the current challenges lie in fixing *semantic bugs*, i.e., cases where the implementation of program behaviour deviates from developer's intention [79, 154].

Ten years ago, the work of Weimer et al. [238] was explicitly motivated by the fact that, despite significant advances in specification mining (e.g., [119]), formal specifications are rarely available. Thus, test suites represented an affordable approximation to program specifications. Unfortunately, the assumption that *test cases are readily available* still does not hold in practice [15, 95, 187]. Therefore, while current test-based APR approaches would be suitable in a test-driven development setting [14], their adoption by practitioners faces a simple reality: developers majoritarily (1) write few tests [95],

(2) write tests after the source code [15], and (3) write tests to validate that bugs are indeed fixed and will not reoccur [80].

Although APR bots [230] can come in handy in a continuous integration environment, the reality is that *bug reports* remain the main source of the stream of bugs that developers struggle to handle daily [11]. Bugs are indeed reported in natural language, where users tentatively describe the execution scenario that was being carried out and the unexpected outcome (e.g., crash stack traces). Such bug reports constitute an essential artefact within a software development cycle and can become an overwhelming concern for maintainers. For example, as early as in 2005, a triager of the Mozilla project was reported in [11, page 363] to have commented that:

> "*Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle.*"

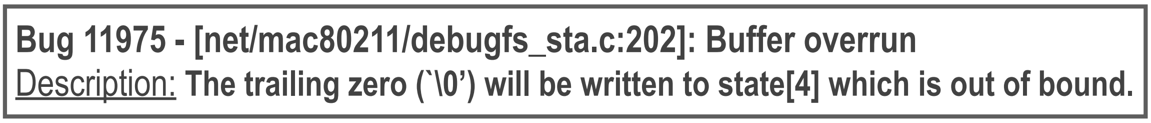
However, a few studies [20, 126] have undertaken to automate patch generation based on bug reports. To the best of our knowledge, Liu et al. [126] proposed the most advanced study in this direction. Their R2Fix approach carries several caveats: as illustrated in Figure 5.8, it focuses on perfect bug reports [126, page 283] (1) which explicitly include localisation information, (2) where the symptom is explicitly indicated by the reporter, and (3) which are about one of the following three simple bug types: Buffer overflow, Null Pointer dereference or memory leak. R2Fix runs a straightforward classification to identify the bug category and uses a match and transform engine to generate patches. As the authors admitted, their target space represents <1% of bug reports in their dataset. It should be noted that, given the limited scope of the changes implemented in its fix patterns, R2Fix does not need to run tests for verifying that the generated patches do not break any functionality.

We propose to investigate the feasibility of a program repair system driven by bug reports, thus we replace classical spectrum-based fault localisation with Information Retrieval (IR)-based fault localisation. Eventually, we propose `iFixR`, a new program repair workflow which considers a practical repair setup by imitating the fundamental steps of manual debugging. `iFixR` works under the following constraint:

When a bug report is submitted to the issue tracking system, a relevant test case reproducing the bug may not be readily available.

Therefore, `iFixR` is leveraged in this study to assess *to what extent an APR pipeline is feasible under the practical constraint of limited test suites.* `iFixR` uses bug reports written in natural language as the main input. Eventually, we make the following contributions:

- We present the architecture of a program repair system adapted to the constraints of maintainers dealing with user-reported bugs. In particular, `iFixR` replaces classical spectrum-based fault localisation with Information Retrieval (IR)-based fault localisation.
- We propose a strategy to prioritise patches for recommendation to developers. Indeed, given that we assume only the presence of regression test cases to validate patch candidates, many of these patches may fail on the future test cases that are relevant to the reported bugs. We order patches to present correct patches first.
- We assess and discuss the performance of `iFixR` on the Defects4J benchmark to compare with the state-of-the-art APR tools. To that end, we provide a refined Defects4J benchmark for APR targeting bug reports. Bugs are carefully linked with the corresponding bug reports, and for each bug we are able to dissociate *future test cases* that were introduced after the relevant fixes.



Bug 11975 - [net/mac80211/debugfs_sta.c:202]: Buffer overrun
Description: **The trailing zero (`\0') will be written to state[4] which is out of bound.**

Figure 5.8: Example of Linux bug report addressed by R2Fix.

Overall, experimental results show that there are promising research directions to further investigate towards the integration of automatic patch generation in actual software development cycles. In particular, our findings suggest that IR-based fault localisation errors lead less to overfitting patches than spectrum-based fault localisation errors. Furthermore, `iFixR` offers comparable results to most state-of-the-art APR tools, although it is run under the constraint that post-fix knowledge (i.e., future test cases) is not available. Finally, `iFixR`'s prioritisation strategy tends to place more correct/plausible patches on top of the recommendation list.

5.10 Motivation

We motivate our work by revisiting two essential steps in APR:

1. During *fault localisation*, relevant program entities are identified as suspicious locations that must be changed. Commonly, state-of-the-art APR tools leverage spectrum-based fault localisation (SBFL) [35,110,133,137,255,257], which uses execution coverage information of passing and failing test cases to predict buggy statements. We dissect the Defects4J dataset to highlight the practical challenges of fault localisation for user-reported bugs.
2. Once a patch candidate is generated, the *patch validation* step ensures that it is actually relevant for repairing the program. Currently, widespread test-based APR techniques use test suites as the repair oracle. This however is challenged by the incompleteness of test suites, and may further not be inline with developer requirements/expectations in the repair process.

5.10.1 Fault Localisation Challenges

Defects4J is a manually curated dataset widely used in the APR literature [35,72,200,241,253,254]. Since Defects4J was not initially built for APR, the real order of precedence between the bug report, the patch and the test case is being overlooked by the dataset users. Indeed, Defects4J offers a user-friendly way of checking out buggy versions of programs with all relevant test cases for readily benchmarking test-based systems. We propose to carefully examine the actual bug fix commits associated with Defects4J bugs and study how the test suite is evolved. Table 5.11 provides detailed information.

Table 5.11: Test case changes in fix commits of Defects4J bugs.

Test case related commits	**# bugs**
Commit does not alter test cases	14
Commit is inserting new test case(s) and updating previous test case(s)	62
Commit is updating previous test case(s) (without inserting new test cases)	76
Commit is inserting new test case(s) (without updating previous test cases)	243

Overall, for 96%(=381/395) bugs, the relevant test cases are actually *future data* with respect to the bug discovery process. This finding suggests that, in practice, even the fault localisation may be challenged in the case of user-reported bugs, given the lack of relevant test cases. The statistics listed in Table 5.12 indeed shows that if future test cases are dropped, no test case is failing when executing buggy program versions for 365 (i.e., 92%) bugs.

In the APR literature, fault localisation is generally performed using the GZoltar [31] testing framework and an SBFL formula [249] (e.g., Ochiai [3]). To support our discussions, we attempt to perform fault localisation without future test cases to evaluate the performance gap. Experimental results (see details forward in Table 5.16 of Section 5.13) expectedly reveal that the majority of the Defects4J bugs (i.e., 375/395) cannot be localised by SBFL at the time the bug is reported by users.

Table 5.12: Failing test cases after removing future test cases.

Failing test cases	**# bugs**
Failing test cases exist (and no future test cases are committed)	14
Failing test cases exist (but future test cases update the test scenarios)	9
Failing test cases exist (but they are fewer when considering future test cases)	4
Failing test cases exist (but they differ from future test cases which trigger the bug)	3
No failing test case exists (i.e., only future test cases trigger the bug)	365

It is necessary to investigate alternate fault localisation approaches that build on bug report information since relevant test cases are often unavailable when users report bugs.

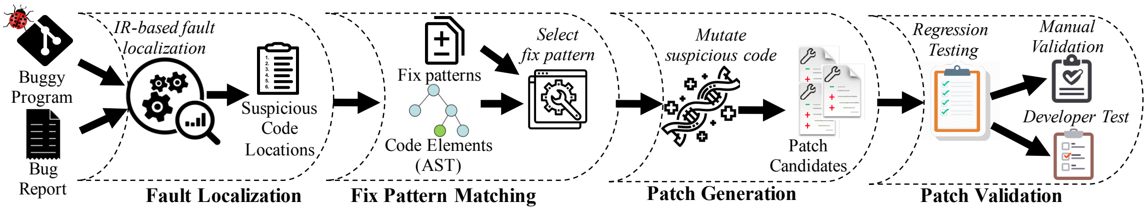


Figure 5.9: The `iFixR` Program Repair Workflow.

5.10.2 Patch Validation in Practice

The repair community has started to reflect on the *acceptability* [88, 162] and *correctness* [210, 254] of the patches generated by APR tools. Notably, various studies [26, 112, 190, 210, 260] have raised concerns about overfitting patches: a typical APR tool that uses a test suite as the correctness criterion can produce a patched program that actually overfits the test-suite (i.e., the patch makes the program pass all test cases but does not actually repair it). Recently, new research directions [252, 269] are being explored in the automation of test case generation for APR to overcome the overfitting issue. Nevertheless, so far they have had minimal positive impact due to the oracle problem [270] in automatic test generation.

At the same time, the software industry takes a more systematic approach for patch validation by developers. For instance, in the open-source community, the Linux development project has integrated a patch generation engine to automate collateral evolutions that are validated by maintainers [97, 177]. In proprietary settings, Facebook has recently reported on their *Getafix* [12] tool, which automatically suggests fixes to their developers. Similarly, Ubisoft developed *Clever* [168] to detect risky commits at commit-time using patterns of programming mistakes from the code history.

Patch recommendation for validation by developers is acceptable in the software development communities. It may thus be worthwhile to focus on tractable techniques for recommending patches in the road to fully automated program repair.

5.11 The `iFixR` Approach

Figure 5.9 overviews the workflow of the proposed `iFixR` approach. Given a defective program, we consider the following issues:

1. **Where is the bug?** We take as input the bug report in the natural language submitted by the program user. We rely on the information in this report to localise the bug positions.

2. **How should we change the code?** We apply fix patterns that are recurrently found in real-world bug fixes. Fix patterns are selected following the structure of the abstract syntax tree representing the code entity of the identified suspicious code.
3. **Which patches are valid?** We make no assumptions on the availability of *positive test cases* [238] that encode functionality requirements at the time the bug is discovered. Nevertheless, we leverage existing test cases to ensure, at least, that the patch does not regress the program.
4. **Which patches do we recommend first?** In the absence of a complete test suite, we cannot guarantee that all patches that pass regression tests will fix the bug. We rely on heuristics to re-prioritise the validated patches in order to increase the probability of placing a correct patch on top of the list.

5.11.1 Input: Bug reports

Issue tracking systems (e.g., Jira) are widely used by software development communities in the open-source and commercial realms. Although they can be used by developers to keep track of the bugs that they encounter and the features to be implemented, issue tracking systems allow for user participation as a communication channel for collecting feedback on software executions in production.

Table 5.13 illustrates a typical bug report when a user of the LANG library code has encountered an issue while using the *NumberUtils* API. A description of erroneous behaviour is provided. Occasionally, the user may include in the bug description some information on how to reproduce the bug. Oftentimes, users simply insert code snippets or dump the execution stack traces.

In this study, among our dataset of 162 bug reports, we note that only 27 (i.e., ∼17%) are reported by users who are also developers contributing to the projects. 15 (i.e., ∼9%) bugs are reported and again fixed by the same project contributors. These percentages suggest that, for the majority of cases, the bug reports are indeed genuinely submitted by users of the software who require project developers' attention.

Table 5.13: Example bug report (Defects4J Lang-7).

Issue No.	LANG-822
Summary	NumberUtils#createNumber - bad behaviour for leading "–"
Description	NumberUtils#createNumber checks for a leading "–" in the string, and returns null if found. This is documented as a work round for a bug in BigDecimal. Returning nulll is contrary to the Javadoc and the behaviour for other methods which would throw NumberFormatException. It's not clear whether the BigDecimal problem still exists with recent versions of Java. However, if it does exist, then the check needs to be done for all invocations of BigDecimal, i.e. needs to be moved to createBigDecimal.

Given the buggy program version and a bug report, `iFixR` must unfold the workflow for precisely identifying (at the statement level) the buggy code locations. We remind the reader that, in this step, future test cases cannot be relied upon. We consider that if such test cases could have triggered the bug, a continuous integration system would have helped developers deal with the bug before the software is shipped towards users.

5.11.2 Fault Localisation w/o Test Cases

To identify buggy code locations within the source code of a program, we resort to Information Retrieval (IR)-based fault localisation (IRFL) [184,231]. The general objective is to leverage potential similarity between the terms used in a bug report and the source code to identify relevant buggy code locations. The literature includes a large body of work on IRFL [143, 199, 232, 242, 247, 268, 276] where researchers systematically extract tokens from a given bug report to formulate a *query* to be matched in a search space of *documents* formed by the collections of source code files and indexed through tokens extracted from source code. IRFL approaches then rank the documents based on a probability of relevance (often measured as a similarity score). Highly ranked files are predicted to be the ones that are likely to contain the buggy code.

Despite recurring interest in the literature, with numerous approaches continuously claiming new performance improvements over the state-of-the-art, we are not aware of any adoption in program repair research or practice. We postulate that one of the reasons is that IRFL techniques have so far focused on file-level localisation, which is too coarse-grained (in comparison to spectrum-based fault localisation output). Recently, Locus [242] and BLIA [268] are state-of-the-art techniques which narrow down localisation, respectively to the code change or the method level. Nevertheless, to the best of our knowledge, no IRFL technique has been proposed in the literature for statement-level localisation.

In this work, we develop an algorithm to rank suspicious statements based on the output (i.e., files) of a state-of-the-art IRFL tool, thus yielding a fine-grained IR-based fault localiser which will then be readily integrated into a concrete patch generation pipeline.

5.11.2.1 Ranking Suspicious Files

We leverage an existing IRFL tool. Given that expensive extraction of tokens from a large corpus of bug reports is often necessary to tune IRFL tools [120], we selected a tool for which the authors provide datasets and preprocessed data. We use the D&C [98] as the specific implementation of file-level IRFL available online [1] , which is a machine learning-based IRFL tool using a similarity matrix of 70-dimension feature vectors (7 features from bug reports and 10 features from source code files): D&C uses multiple classifier models that are trained each for specific groups of bug reports. Given a bug report, the different predictions of the different classifiers are merged to yield a single list of suspicious code files. Our execution of D&C (Line 2 in Algorithm 4) is tractable given that we only need to preprocess those bug reports that we must localise. Trained classifiers are already available. We ensure that no data leakage is induced (i.e., the classifiers are not trained with bug reports that we want to localise in this work).

5.11.2.2 Ranking Suspicious Statements

Patch generation requires fine-grained information on code entities that must be changed. For `iFixR`, we propose to produce a standard output, as for spectrum-based fault localisation, to facilitate integration and reuse of state-of-the-art patch generation techniques. To start, we build on the conclusions on a recent large-scale study [130] of bug fixes to limit the search space of suspicious locations to the statements that are more error-prone. After investigating in detail the abstract syntax tree (AST)-based code differences of over 16 000 real-world patches from Java projects, Liu et al. [130] reported that the following specific AST statement nodes were significantly more prone to be faulty than others: `IfStatements`, `ExpressionStatements`, `FieldDeclarations`, `ReturnStatements` and `VariableDeclarationStatements`. Lines 7–17 in Algorithm 4 detail the process to produce a ranked list of suspicious statements.

Algorithm 4 describes the process of our fault localisation approach used in `iFixR`. Top k files are selected among the returned list of suspicious files of the IRFL along with their computed suspiciousness scores. Then each file is parsed to retain only the relevant error-prone statements from which textual tokens are extracted. The summary and descriptions of the bug report are also analysed (lexically) to collect all its tokens. Due to the specific nature of stack traces and other code elements which may appear in the bug report, we use regular expressions to detect stack traces

Algorithm 4: Statement-level IR-based Fault Localization.

```
Input    : $br$ : a bug report
Input    : $irTool$ : IRFL tool
Output   : $S_{score}$ : Suspicious Statements with weight scores
Function main ($br$,$irTool$)
    $F \leftarrow$ fileLocalizations ($irTool$,$br$)
    $F \leftarrow$ selectTop ($F$,$k$)
    $c_b \leftarrow$ bagOfTokens ($br$)                 /* $c_b$: Bag of Tokens of bug report */
    $c'_b \leftarrow$ preprocess ($c_b$)        /* tokenization,stopword removal, stemming */
    $v_b \leftarrow$ tfIdfVectorizer($c'_b$)              /* $v_b$: Bug report Feature Vector */
    for $f$ in $F$ do
        $S \leftarrow$ parse($f$)                                /* $S$: List of statements */
        for $s$ in $S$ do
            $c_s \leftarrow$ bagOfTokens ($s$)          /* $c_s$: Bag of Tokens of statements */
            $c'_s \leftarrow$ preprocess ($c_s$)
            $v_s \leftarrow$ tfIdfVectorizer($c'_s$)        /* $v_s$: Statements Feature Vector */
            /* Cosine similarity between bug report and statement                  */
            $sim_{cos} \leftarrow$ similarity$_{\texttt{cosine}}$ ($v_b$,$v_s$)
            $w_{score} \leftarrow sim_{cos} \times f$.score;            /* score: Suspicious Value */
            $W_{score}$.add($s$,$w_{score}$)
    $S_{score} \leftarrow W_{score}$.sort()
    return $S_{score}$
```

and code elements to improve the tokenisation process, which is based on punctuations, camel case splitting (e.g., findNumber splits into find, number) as well as snake case splitting (e.g., find_number splits into find, number). Stop word removal [2] is then applied before performing stemming (using the PorterStemmer [82]) on all tokens to create homogeneity with the term's root (i.e., by conflating variants of the same term). Each bag of tokens (for the bug report, and for each statement) is then eventually used to build a feature vector. We use cosine similarity among the vectors to rank the file statements that are relevant to the bug report.

Given that we considered k files, the statements of each having their own similarity score with respect to the bug report, we weight these scores with the suspiciousness score of the associated file. Eventually, we sort the statements using the weighted scores and produce a ranked list of code locations (i.e., statements in files) to be recommended as candidate fault locations.

5.11.3 Fix Pattern-based Patch Generation

A common, and reliable, strategy in automatic program repair is to generate concrete patches based on fix patterns [88] (also referred to as fix templates [135] or program transformation schemas [72]). Several APR systems [48,72,88,99,129,131,132,135,153,200] in the literature implement this strategy by using diverse sets of fix patterns obtained either via manual generation or automatic mining of bug fix datasets. In this work, we consider the pioneer *PAR* system by Kim et al. [88]. Concretely, we build on *kPAR* [131], an open-source Java implementation of *PAR* in which we included a diverse set of fix patterns collected from the literature. Table 5.14 provides an enumeration of fix patterns used in this work. For more implementation details, we refer the reader to our replication package. All tools and data are released as open-source to the community to foster further research into these directions. As illustrated in Figure 5.10, a fix pattern encodes the recipe of change actions that should be applied to mutate a code element.

For a given reported bug, once our fault localiser yields its list of suspicious statements, `iFixR` iteratively attempts to select fix patterns for each statement. The selection of fix patterns is

Table 5.14: Fix patterns implemented in iFixR.

Pattern description	used by*	Pattern description	used by*
Insert Cast Checker	Genesis	Mutate Literal Expression	SimFix
Insert Null Pointer Checker	NPEFix	Mutate Method Invocation	ELIXIR
Insert Range Checker	SOFix	Mutate Operator	jMutRepair
Insert Missed Statement	HDRepair	Mutate Return Statement	SketchFix
Mutate Conditional Expression	ssFix	Mutate Variable	CapGen
Mutate Data Type	AVATAR	Move Statement(s)	PAR
Remove Statement(s)	FixMiner		

* We mention only one example tool even when several tools implement it.

```
+ if (exp instanceof T) {
      ...(T) exp...; ......
+ }
```

Figure 5.10: Illustration of "Insert Cast Checker" fix pattern.

```
File: src/main/java/org/apache/commons/math/stat/Frequency.java
Line—301    public double getPct(Object v) {
Line—302            return getCumPct((Comparable<?>) v);
Line—303    }
```

Figure 5.11: Buggy code of Defects4J bug Math-75.

conducted in a naïve way based on the context information of each suspicious statement (i.e., all nodes in its abstract syntax tree, AST). Specifically, iFixR parses the code and traverses each node of the suspicious statement AST from its first child node to its last leaf node in a breadth-first strategy (i.e., left-to-right and top-to-bottom). If a node matches the context a fix pattern (i.e., same AST node types), the fix pattern will be applied to generate patch candidates by mutating the matched code entity following the recipe in the fix pattern. Whether the node matches a fix pattern or not, iFixR keeps traversing its children nodes and searches fix patterns for them to generate patch candidates successively. This process is iteratively performed until leaf nodes are encountered.

Consider the example of bug Math-75 illustrated in Figure 5.11. iFixR parses the buggy statement (i.e., statement at line 302 in the file *Frequency.java*) into an AST as illustrated by Figure 5.12. First, iFixR matches a fix pattern that can mutate the expression in the return statement with other expression(s) returning data of type *double*. It further selects fix patterns for the direct child node (i.e., method invocation: `getCumPct((Comparable<?> v))`) of the return statement. This method invocation can be matched against fix patterns with two contexts: method name and parameter(s). With the breadth-first strategy, iFixR assigns a fix pattern, calling another method with the same parameters (cf. PAR [88, page 804]), to mutate the method name, and then selects fix patterns to mutate the parameter. Furthermore, iFixR will match fix patterns for the type and variable of the cast expression respectively and successively.

5.11.4 Patch Validation with Regression Testing

For every reported bug, fault localisation followed by pattern matching and code mutation will yield a set of patch candidates. In a typical test-based APR system, these patch candidates must let the program pass all test cases (including some *positive test cases* [238], which encode the actual functional requirements relevant to the bug). Thus, the patch candidates set is actively pruned to remove all patches that do not meet these requirements. In our work, in accordance with our investigation findings that such test cases may not be available at the time the bug is reported (cf. Section 5.10), we assume that iFixR cannot reason about *future* test cases to select patch candidates.

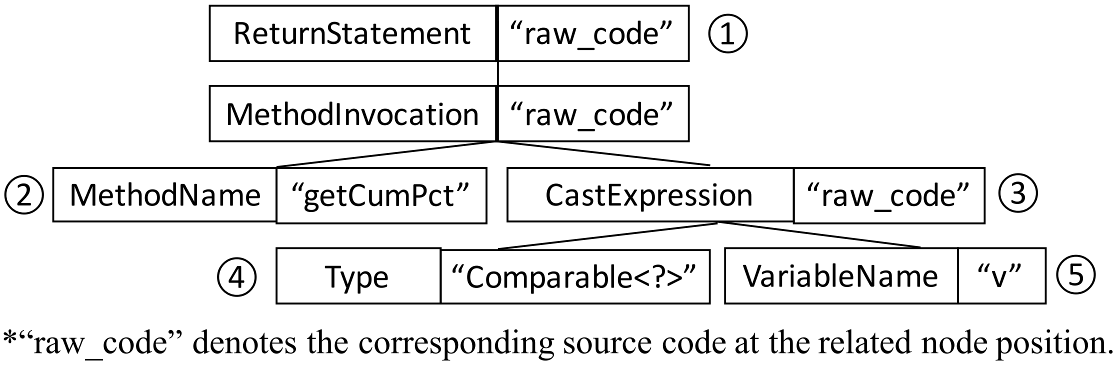


Figure 5.12: AST of bug Math-75 source code statement.

Instead, we rely only on *past* test cases, which were available in the code base, when the bug is reported. Such test cases are leveraged to perform *regression testing* [266], which will ensure that, at least, the selected patches do not obstruct the behaviour of the existing, unchanged part of the software, which is already explicitly encoded by developers in their current test suite.

5.11.5 Output: Patch Recommendation List

Eventually, `iFixR` produces a ranked recommendation list of patch suggestions for developers. Until now, the order of patches is influenced mainly by two steps in the workflow:

1. localisation: our statement-level IRFL yields a ranked list of statements to modify in priority.
2. pattern matching: the AST node of the buggy code entity is broken down into its children and iteratively navigated in a breadth-first manner to successively produce candidate patches.

Eventually, the produced list of patches has an order, which carries the biases of fault localisation [131], and is noised by the pre-set breadth-first strategy for matching fix patterns. We thus design an ordering process with a function[8], $f_{rcmd} : 2^{\mathbb{P}} \to \mathbb{P}^k$, as follows:

$$f_{rcmd}(patches) = (pri_{type} \circ pri_{susp} \circ pri_{change})(patches) \tag{5.13}$$

where pri_* are three heuristics-based prioritisation functions used in `iFixR`. f_{rcmd} takes a set of patches validated via regression testing (cf. Section 5.11.4) and produces an ordered sequence of patches ($f_{rcmd}(patches) = seq_{rcmd} \in \mathbb{P}^k$). We propose the following **heuristics to re-prioritise the patch candidates**:

1. `[Minimal changes]`: we favour patches that minimize the differences between the patched program and the buggy program. To that end, patches are ordered following their AST edit script sizes. Formally, we define $pri_{change} : 2^{\mathbb{P}} \to \mathbb{P}^n$ where $n = |patches|$, $pri_{change}(patches) = [p_i, p_{i+1}, p_{i+2}, \cdots]$ and holds $\forall p \in patches, C_{change}(p_i) \leq C_{change}(p_{i+1})$. Here, $C_{change}(p)$ is a function that counts the number of deleted and inserted AST nodes by the change actions of p.
2. `[Fault localisation suspiciousness]`: when two patch candidates have equal edit script sizes, the tie is broken by using the suspiciousness scores (of the associated statements) yielded during IR-based fault localisation. Thus, when $C_{change}(p_i) == C_{change}(p_{i+1})$, pri_{susp} re-orders the two patch candidates. We define $pri_{susp} : \mathbb{P}^n \to \mathbb{P}^n$ such that $pri_{susp}(seq_{change}) = [\cdots, p_i, p_{i+1}, \cdots]$ holds $S_{susp}(p_i) \geq S_{susp}(p_{i+1})$, where
seq_{change} is the result of pri_{change} and S_{susp} returns a suspicious score of the statement that a given patch p_i changes.

[8] The domain of the function is a power set $2^{\mathbb{P}}$, and the co-domain ($\mathbb{P}^k$) is a k-dimensional vector space [96] where k is the maximum number of recommended patches, and $\mathbb{P}$ denotes the set of all generated patches.

3. `[Affected code elements]`: after a manual analysis of fix patterns and the performance of associated APR in the literature, we empirically found that some change actions are irrelevant to bug fixing. Thus, for the corresponding pre-defined patterns, `iFixR` systematically under-prioritises their generated patches against any other patches, although among themselves the ranking obtained so far (through pri_{change} and pri_{susp}) is preserved for those under-prioritised patches. These are patches generated by (i) mutating a literal expression, (ii) mutating a variable into a method invocation or a final static variable, or (iii) inserting a method invocation without parameter. This prioritisation, is defined by $pri_{type} : \mathbb{P}^n \rightarrow \mathbb{P}^k$, which returns a sequence of top k ordered patches ($k \leq n = |patches|$). To define this prioritisation function, we assign natural numbers $j_1, j_2, j_3, j_4 \in \mathbb{N}$ to each patch generation types (i.e., $j_1 \leftarrow$(i), $j_2 \leftarrow$(ii), and $j_3 \leftarrow$(iii), respectively) and ($j_4 \leftarrow$) everything else, which strictly hold $j_4 > j_1, j_4 > j_2, j_4 > j_3$. This prioritisation function takes the result of pri_{susp} and returns another sequence $[p_i, p_{i+1}, p_{i+2}, \cdots]$ that holds $\forall p_i, D_{type}(p_i) \geq D_{type}(p_{i+1})$. D_{type} is defined as $D_{type} : 2^{\mathbb{P}} \rightarrow \{j_1, j_2, j_3, j_4\}$ and determines how a patch p_i has been generated as defined above. From the ordered sequence, the function returns the leftmost (i.e., top) k patches as a result.

5.12 Experimental Setup

We now provide details on the experiments that we carry out to assess the `iFixR` patch generation pipeline for user-reported bugs. Notably, we discuss the dataset and benchmark, some implementation details before enumerating the research questions.

5.12.1 Dataset & Benchmark

To evaluate `iFixR` we rely on the Defects4J [78] which is widely used as a benchmark in the Java APR literature. Nevertheless, given that Defects4J does not provide direct links to the bug reports that are associated with the benchmark bugs, we must undertake a *fairly accurate* bug linking task [223]. Furthermore, to realistically evaluate `iFixR`, we reorganise the dataset test suites to accurately simulate the context at the time the bug report is submitted by users.

5.12.1.1 Bug linking

To identify the bug report describing a given bug in the Defects4J dataset we focus on recovering the links between the bug fix commits and bug reports from the issue tracking system. Unfortunately, projects Joda-Time, JFreeChart and Closure have migrated their source code repositories and issue tracking systems into GitHub without a proper reassignment of bug report identifiers. Therefore, for these projects, bug IDs referred to in the commit logs are ambiguous (for some bugs this may match with the GitHub issue tracking numbering, while in others, it refers to the original issue tracker). To avoid introducing noise in our validation data, we simply drop these projects. For the remaining projects (Lang and Math), we leverage the bug linking strategies implemented in the Jira issue tracking software. We use a similar approach to Fischer et al. [53] and Thomas et al. [223] to link to commits to corresponding bug reports. Concretely, we crawled the bug reports related to each project and assessed the links with a two-step search strategy: (i) we check commit logs to identify bug report IDs and associate the corresponding changes as bug fix changes; then (ii) we check for bug reports that are indeed considered as such (i.e., tagged as "BUG") and are further marked as resolved (i.e., with tags "RESOLVED" or "FIXED"), and completed (i.e., with status "CLOSED").

Eventually, our evaluation dataset includes **156 faults** (i.e., Defects4J bugs). Actually, for the considered projects, Defects4J enumerates 171 bugs associated with **162 bug reports**: 15 bugs are indeed left out because either (1) the corresponding bug reports are not in the desired status in the bug tracking system, which may lead to noisy data, or (2) there is ambiguity in the buggy program version (e.g., some fixed files appear to be missing in the repository at the time of bug reporting).

5.12.1.2 Test suite reorganisation

We ensure that the benchmark separates past test cases (i.e., regression test cases) from future test cases (i.e., test cases that encode functional requirements specified after the bug is reported). This timeline split is necessary to simulate the snapshot of the repository at the time the bug is reported. As highlighted in Section 5.10, for over 90% cases of bugs in the Defects4J benchmark, the test cases relevant to the defective behaviour was actually provided along the bug fixing patches. We have thus manually split the commits to identify test cases that should be considered as future test cases for each bug report.

5.12.2 Implementation Choices

During implementation, we have made the following parameter choices in the `iFixR` workflow:

- IR fault localisation considers the top 50 (i.e., $k = 50$ in Algorithm 4) suspicious files for each bug report, in order to search for buggy code locations.
- For patch recommendation experiments, we limit the search space to the top 20 suspected buggy statements yielded by the fine-grained IR-based fault localisation.
- For comparison experiments, we implement spectrum-based fault localisation using the GZoltar testing framework with the Ochiai ranking strategy. Unless otherwise indicated, GZoltar version 0.1.1 is used (as it is widely adopted in the literature, by Astor [152], ACS [255], ssFix [253] and CapGen [241] among others).

5.12.3 Research Questions

The assessment objective is to assess the **feasibility of automating the generation of patches for user-reported bugs**, while investigating the foreseen bottlenecks as well as the research directions that the community must embrace to realize this long-standing endeavour. To that end, we focus on the following research questions associated with the different steps in the `iFixR` workflow.

- RQ1 [Fault localisation] : *To what extent does IR-based fault localisation provide reliable results for an APR scenario?* In particular, we investigate the performance differences when comparing our fine-grained IRFL implementation against the classical spectrum-based localisation.
- RQ2 [Overfitting] : *To what extent does IR-based fault localisation point to locations that are less subject to overfitting?* In particular, we study the impact on the *overfitting* problem that incomplete test suites generally carry.
- RQ3 [Patch ordering] : *What is the effectiveness of* ***`iFixR`****'s patch ordering strategy?* In particular, we investigate the overall workflow of `iFixR`, by re-simulating the real-world cases of software maintenance cycle when a bug is reported: future test cases are not available for patch validation.

5.13 Assessment Results

In this section, we present the results of the investigations for the previously-enumerated research questions.

5.13.1 RQ1: [Fault Localisation]

Fault localisation being the first step in program repair, we evaluate the performance of the IR-based fault localisation developed within `iFixR`. As recently thoroughly studied by Liu et al. [131], an APR tool should not be expected to fix a bug that current fault localisation systems fail to localise. Nevertheless, with `iFixR`, we must demonstrate that our fine-grained IRFL offers comparable performance with SBFL tools used in the APR literature.

Table 5.15 provides performance measurements on the localisation of bugs. SBFL is performed based on two different versions of the GZoltar testing framework, but always based on the Ochiai ranking metric. Finally, because fault localisation tools output a ranked list of suspicious statements, results are provided in terms of whether the correct location is placed under the top-k suspected statements. In this work, following the practice in the literature [131, 142], we consider that a bug is localised if any buggy statement is localised.

Table 5.15: Fault localization results: IRFL (IR-based) vs. SBFL (Spectrum-based) on Defects4J (Math and Lang) bugs.

(171 bugs)		Top-1	Top-10	Top-50	Top-100	Top-200	All
IRFL		25	72	102	117	121	139
SBFL	**GZ**$_{v1}$	26	75	106	110	114	120
	GZ$_{v2}$	23	79	119	135	150	156

† GZ$_{v1}$ and GZ$_{v2}$ refer to GZoltar 0.1.1 and 1.6.0 respectively.

Overall, the results show that our IRFL implementation is strictly comparable to the common implementation of spectrum-based fault localisation when applied on the Defects4J bug dataset. Note that the comparison is conducted for 171 bugs of Math and Lang, given that these are the projects for which the bug linking can be reliably performed for applying the IRFL. Although performance results are similar, we remind the reader that SBFL is applied by considering future test cases. To highlight a practical interest of IRFL, we compute for each bug localisable in the top-10, the elapsed time between the bug report date and the date the relevant test case is submitted for this bug. Based on the distribution shown in Figure 5.13, on mean average, IRFL could reduce this time by 26 days.

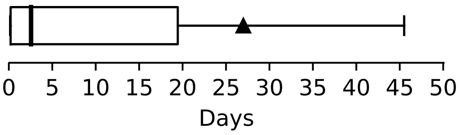


Figure 5.13: Distribution of elapsed time (in days) between bug report submission and test case attachment.

Finally, to stress the importance of future test cases for spectrum-based fault localisation, we consider all Defects4J bugs and compute localisation performance with and without future test cases.

Results listed in Table 5.16 confirms that in most bug cases, the localisation is impossible: Only 10 bugs (out of 395) can be localised among the top-10 suspicious statements of SBFL at the time the bug is reported. In comparison, our IRFL locates 72 bugs under the same conditions of having no relevant test cases to trigger the bugs.

> *Fine-grained IR-based fault localisation in `iFixR` is as accurate as Spectrum-based fault localisation in localising Defects4J bugs. Additionally, it does not have the constraint of requiring test cases that may not be available when the bug is reported.*

Table 5.16: Fault localization performance.

GZoltar + Ochiai (395 bugs)	Top-1	Top-10	Top-50	Top-100	Top-200	All
without future tests	5	10	17	17	19	20
with future tests	45	140	198	214	239	263

5.13.2 RQ2: [Overfitting]

Patch generation attempts to mutate suspected buggy code with suitable fix patterns. Aside from having adequate patterns or not (which is out of the scope of our study), a common challenge of APR lies in the effective selection of buggy statements. In typical test-based APR, test cases drive the selection of these statements. The incompleteness of test suites is however currently suspected to often lead to overfitting of generated patches [260].

We perform patch generation experiments to investigate the impact of localisation bias. We compare our IRFL implementation against commonly-used SBFL implementations in the literature of test-based APR. We recall that the patch validation step in these experiments makes no assumptions about future test cases (i.e., all test cases are leveraged as in classical APR pipeline). For each bug, depending on the rank of the buggy statements in the suspicious statements yielded the fault localisation system (either IRFL or SBFL), the patch generation can produce more or less relevant patches. Table 5.17 details the repair performance in relation to the position of buggy statements in the output of fault localisation. Results are provided in terms of numbers of *plausible* and *correct* [190] patches that can be found by considering top-k statements returned by the fault localiser.

Table 5.17: IRFL vs. SBFL impacts on the number of generated correct/plausible patches for Defects4J bugs.

	Lang	**Math**	**Total**
IRFL Top-1	1/4	3/4	4/8
SBFL Top-1	1/4	**6/8**	7/12
IRFL Top-5	**3**/6	7/14	10/20
SBFL Top-5	2/**7**	**11/17**	13/24
IRFL Top-10	**4/9**	9/17	13/26
SBFL Top-10	4/11	**16/27**	20/38
IRFL Top-20	**7/12**	9/18	16/30
SBFL Top-20	4/11	**18/30**	22/41
IRFL Top-50	**7/15**	10/22	17/37
SBFL Top-50	4/13	**19/34**	23/47
IRFL Top-100	**8/18**	10/23	18/41
SBFL Top-100	5/14	**19/36**	24/50
IRFL All	**11/19**	10/25	21/44
SBFL All	5/14	**19/36**	24/50

* We indicate x/y numbers of patches: x is the number of bugs for which a *correct* patch is generated; y is the number of bugs for which a *plausible* patch is generated.

Overall, we find that IRFL and SBFL localisation information lead to similar repair performance in terms of the number of fixed bugs plausibly/correctly. Actually, IRFL-supported APR outperforms SBFL-supported APR on the Lang project bugs and vice-versa for Math project bugs: overall, 6 bugs that are fixed using IRFL output, cannot be fixed using SBFL output (although assuming the availability of the bug triggering test cases to run the SBFL tool).

We investigate the cases of plausible patches in both localisation scenarios to characterize the reasons why these patches appear to only be overfitting the test suites. Table 5.18 details the overfitting reasons for the two scenarios.

Table 5.18: Dissection of reasons why patches are plausible* but not correct.

	Localization Error	Pattern Prioritization	Lack of Fix ingredients
w/ IRFL	6	1	16
w/ SBFL	15	1	10

*A plausible patch passes all test cases, but may not be semantically equivalent to developer patch (i.e., correct). We consider a plausible patch to be overfitted to the test suite

1. Among the 23(= 44 – 21) plausible patches that are generated based on IRFL identified code locations and that are not found to be correct, 6 are found to be caused by fault localisation errors: these bugs are plausibly fixed by mutating irrelevantly-suspicious statements that are placed before the actual buggy statements in the fault localisation output list. This phenomenon has been recently investigated in the literature as the problem of fault localisation bias [131]. Nevertheless, we note that patches generated based on SBFL identified code locations suffer more of fault localisation bias: 15 of the 26 (= 50−24) plausible patches are concerned by this issue.
2. Pattern prioritisation failures may lead to plausible patches: while a correct patch could have been generated using a specific pattern at a lower node in the AST, another pattern (leading to an only plausible patch) was first matched the node during the iterative search of matching nodes (cf. Section 5.11.3).
3. Finally, we note that both configurations yield plausible patches due to the lack of suitable patterns or due to a failed search for the adequate donor code (i.e., fix ingredient [127]).

> *Experiments with the Defects4J dataset suggest that code locations provided by IR-based fault localisation lead less to overfitted patches than the code locations suggested by Spectrum-based fault localisation: cf. "Localisation error" column in Table 5.18.*

5.13.3 RQ3: [Patch Ordering]

While the previous experiment focused on patch generation, our final experiment assesses the complete pipeline of `iFixR` as it was imagined for meeting the constraints that developers can face in practice: future test cases, i.e., those which encode the functionality requirements that are not met by the buggy programs, may not be available at the time the bug is reported. We thus discard the future test cases of the Defects4J dataset and generate patches that must be recommended to developers. The evaluation protocol thus consists in assessing to what extent correct/plausible patches are placed in the top of the recommendation list.

5.13.3.1 Overall performance

Table 5.19 details the performance of the patch recommendation by `iFixR`: we present the number of bugs for which a correct/plausible patch is generated and presented among the top-k of the list of recommended patches. In the absence of future test cases to drive the patch validation process, we use heuristics (cf. Section 5.12.2) to re-prioritise the patch candidates towards ensuring that patches which are recommended first will eventually be correct (or at least plausible when relevant test cases are implemented). We present results both for the case where we do not re-prioritise and the case where we re-prioritise.

Recall that, given that the re-organised benchmark separately includes the future test cases, we can leverage them to systematise the assessment of patch plausibility. The *correctness* (also referred to as *correctness* [190]) of patches, however, is still decided manually by comparing against the actual bug fix provided by developers and available in the benchmark. Overall, we note that `iFixR` performance is promising as it manages, for **13 bugs**, to present a plausible patch among its top-5 recommended patches per bug. Among those plausible patches, 8 are eventually found to be correct.

Table 5.19: Overall performance of `iFixR` for patch recommendation on the Defects4J benchmark.

Recommendation rank	Top-1	Top-5	Top-10	Top-20	All
without patch re-prioritization	3/3	4/5	6/10	6/10	13/27
with patch re-prioritization	3/4	8/13	9/14	10/15	13/27

* x/y: x is the number of bugs for which a *correct* patch is generated; y is the number of bugs for which a *plausible* patch is generated.

5.13.3.2 Comparison with the state-of-the-art test-based APR systems

To objectively position the performance of `iFixR` (which does not require future test cases to localise bugs, generate patches and present a sorted recommendation list of patches), we count the number of bugs for which `iFixR` can propose a correct/plausible patch. We consider three scenarios with `iFixR`:

1. [`iFixR`$_{top5}$] - developers will be provided with only top 5 recommended patches which have been validated only with regression tests: in this case, `iFixR` outperforms about half of the state-of-the-art in terms of numbers bugs fixed with both plausible or correct patches.
2. [`iFixR`$_{all}$] - developers are presented with all (i.e., not only top-5) generated patches validated with regression tests: in this case, only four (out of sixteen) state-of-the-art APR techniques outperform `iFixR`.
3. [`iFixR`$_{opt}$] - developers are presented with all generated patches which have been validated with augmented test suites (i.e., optimistically with future test cases): with this configuration, `iFixR` outperforms all state-of-the-art, except SimFix [76] which uses sophisticated techniques to improve the fault localisation accuracy and search for fix ingredients. It should be noted that in this case, our prioritisation strategy is not applied to the generated patches. `iFixR`$_{opt}$ represents the reference performance for our experiment which assesses the prioritisation.

Table 5.20 provides the comparison matrix. Information on state-of-the-art results is excerpted from their respective publications.

> *`iFixR` offers reasonable performance in patch recommendation when we consider the number of Defects4J bugs that are successfully patched among the top-5 (in a scenario where we assume not having relevant test cases to validate the patch candidates). Performance results are even comparable to many state-of-the-art test-based APR tools in the literature.*

5.13.3.3 Properties of `iFixR`'s patches

In Table 5.21, we characterise the correct and plausible patches recommended by `iFixR`$_{top5}$. Overall, update and insert changes have been successful; most patches affect a single statement, and impact precisely an expression entity within a statement.

5.13.3.4 Diversity of `iFixR`'s fixed bugs

Finally, in Table 5.22 we dissect the nature of the bugs for which `iFixR`$_{top5}$ is able to recommend a correct or a plausible patch. Priority information about the bug report is collected from the issue tracking systems, while the root cause is inferred by analysing the bug reports and fixes.

Table 5.20: `iFixR` vs state-of-the-art APR tools.

APR tool	**Lang***	**Math***	**Total***
jGenProg [152]	0/0	5/18	5/18
jKali [152]	0/0	1/14	1/14
jMutRepair [152]	0/1	2/11	2/12
HDRepair [114]	2/6	4/7	6/13
Nopol [257]	3/7	1/21	4/28
ACS [255]	3/4	12/16	15/20
ELIXIR [200]	8/12	12/19	20/31
JAID [35]	1/8	1/8	2/16
ssFix [253]	5/12	10/26	15/38
CapGen [241]	5/5	12/16	17/21
SketchFix [72]	3/4	7/8	10/12
FixMiner [99]	2/3	12/14	14/17
LSRepair [127]	8/14	7/14	15/28
SimFix [76]	9/13	**14/26**	**23**/39
kPAR [131]	1/8	7/18	8/26
AVATAR [132]	5/11	6/13	11/24
`iFixR`$_{opt}$	**11/19**	10/25	21/**44**
`iFixR`$_{all}$	6/11	7/16	13/27
`iFixR`$_{top5}$	3/7	5/6	8/13

* *x/y*: x is the number of bugs for which a *correct* patch is generated; y is the number of bugs for which a *plausible* patch is generated.
`iFixR`$_{opt}$: the version of `iFixR` where available test cases are relevant to the bugs.
`iFixR`$_{all}$: all recommended patches are considered.
`iFixR`$_{top5}$: only top 5 recommended patches are considered.

Table 5.21: Change properties of `iFixR`'s correct patches.

Change action	#bugs*	Impacted statement(s)	#bugs*	Granularity	#bugs*
Update	5/7	Single-statement	8/12	Statement	1/2
Insert	3/5	Multiple-statement	0/1	Expression	7/11
Delete	0/1				

* x/y ⟶ for x bugs the patches are correct, while for y bugs they are plausible.

Table 5.22: Dissection of bugs successfully fixed by `iFixR`.

Patch Type	Defects4J Bug ID	Issue ID	Root Cause	Priority
G	L-6	LANG-857	String index out of bounds exception	Minor
G	L-24	LANG-664	Wrong behaviour due missing condition	Major
G	L-57	LANG-304	Null pointer exception	Major
G	M-15	MATH-904	Double precision floating point format error	Major
G	M-34	MATH-779	Missing "read only access" to internal list	Major
G	M-35	MATH-776	Range check	Major
G	M-57	MATH-546	Wrong variable type truncates double value	Minor
G	M-75	MATH-329	Method signature mismatch	Minor
P	L-13	LANG-788	Serialization error in primitive types	Major
P	L-21	LANG-677	Wrong Date Format in comparison	Major
P	L-45	LANG-419	Range check	Minor
P	L-58	LANG-300	Number formatting error	Major
P	M-2	MATH-1021	Integer overflow	Major

"G" denotes correct patch and "P" means plausible patch.

Overall, we note that 9 out of the 13 bugs have been marked as Major issues. 12 different bug types (i.e., root causes) are addressed. In contrast, R2Fix [126] only focused on 3 simple bug types.

5.14 Discussion

This study presents the conclusions of our investigation into the feasibility of generating patches automatically from bug reports. We set strong constraints on the absence of test cases, which are used in test-based APR to approximate *what the program is actually supposed to do* and *when the repair is completed* [238]. Our experiments on the widely-used Defects4J bugs eventually show that *patch generation without bug-triggering test cases* is promising.

Manually looking at the details of failures and success in generating patches with iFixR, several insights can be drawn:

Test cases can be buggy: During the manual analysis of results, we noted that iFixR actually fails to generate correct patches for three bugs (namely, Math-5, Math-59 and Math-65) because even the test cases were buggy. Figure 5.14 illustrates the example of bug Math-5, where its patch also updated the relevant test case. This example supports our endeavour, given that users would find and report bugs for which the appropriate test cases were never properly written.

```
// Patched Source Code:
−−− a/src/main/java/org/apache/commons/math3/complex/Complex.java
+++ b/src/main/java/org/apache/commons/math3/complex/Complex.java
@@ −304,3 +304,3 @@ public class Complex
        if (real == 0.0 && imaginary == 0.0) {
-           return NaN;
+           return INF;
        }
// Patched Test Case:
— a/src/test/java/org/apache/commons/math3/complex/ComplexTest.java
+++ b/src/test/java/org/apache/commons/math3/complex/ComplexTest.java
@@ −333,3 +333,3 @@ public class ComplexTest {
    public void testReciprocalZero() {
-       Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.NaN);
+       Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.INF);
    }
```

Figure 5.14: Patched source code and test case of fixing Math-5.

Bug reports deserve more interest: With iFixR, we have shown that bug reports could be handled automatically for a variety of bugs. This is an opportunity for issue trackers to add a recommendation layer to the bug triaging process by integrating patch generation techniques. There are, however, several directions to further investigation, among which: (1) help users write proper bug reports; and (2) re-investigate IRFL techniques at a finer-grained level that is suitable for APR.

Prioritisation techniques must be investigated: In the absence of complete test suites for validating every single patch candidate, a recommendation system must ensure that patches presented first to the developers are the most likely to be plausible and even correct. There are thus two directions of research that are promising: (1) ensure that fix patterns are properly prioritised to generate good patches and be able to early-stop for not exploding the search space; and (2) ensure that candidate patches are effectively re-prioritised. These investigations must start with a thorough dissection of plausible patches for a deep understanding of plausibility factors.

More sophisticated approaches to triaging and selecting fix ingredients are necessary: In its current form, iFixR implements a naïve approach to patch generation, ensuring that the performance is tractable. However, the literature already includes novel APR techniques that implement strategies for selecting donor code and filters patterns. Integrating such techniques into iFixR may lead to performance improvement.

More comprehensive benchmarks are needed: Due to bug linking challenges, our experiments were only performed on half of the Defects4J benchmark. To drive strong research in patch generation for user-reported bugs, the community must build larger and reliable benchmarks, potentially even linking several artefacts of continuous integration (i.e., build logs, past execution traces, etc.). In the future, we plan to investigate the dataset of Bugs.jar [197].

Automatic test generation techniques could be used as a supplement: Our study tries to cope radically with the incompleteness of test suites. In the future, however, we could investigate the use of automatic test generation techniques to supplement the regression test cases during patch validation.

5.15 Threats to Validity

Threats to external validity: The bug reports used in this study may be of low quality (i.e., wrong links for corresponding bugs). We reduced this threat by focusing only on bugs from the Lang and Math projects, which kept a single issue tracking system. We also manually verified the links between the bug reports and the Defects4J bugs. Table 5.23 characterises the bug reports of our dataset following the criteria enumerated by Zimmermann et al. [277] in their study of “what makes a good bug report”. Notably, as illustrated by the distribution of comments in Figure 5.15, we note that the bug reports have been actively discussed before being resolved. This suggests that they are not trivial cases (cf. [69]).

Table 5.23: Dissection of bug reports related to Defects4J bugs.

Proj.	Unique Bug Reports	w/ Patch Attached	Average Comments	w/ Stack Traces	w/ Hints	w/ Code Blocks
Lang	62	11	4.53	4	62	31
Math	100	23	5.15	5	92	51

Code-related terms such as package/class names found in the summary and description, in addition to stack traces and code blocks, as separate features referred to as hints.

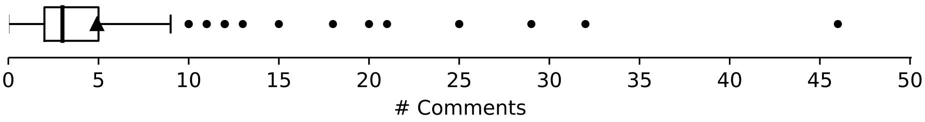


Figure 5.15: Distribution of # of comments per bug report.

Another threat to external validity relates to the diversity of the fix patterns used in this study. iFixR currently may not implement a reasonable number of relevant fix patterns. We minimise this threat by surveying the literature and considering patterns from several pattern-based APR.

Threats to internal validity: Our implementation of fine-grained IRFL carries some threats: during the search of buggy statements, we considered top-50 suspicious buggy files from the file-level IRFL tool, to limit the search space. Different threshold values may lead to different results. We also considered only 5 statement types as more bug-prone. This second threat is minimised by the empirical evidence provided by Liu et al. [130].

Additionally, another internal threat is in our patch generation steps: iFixR only searches for donor code from the local code files, which contain the buggy statement. The adequate fix ingredient may however be located elsewhere.

Threats to construct validity: In this study, we assumed that patch construction and test case creation are two separated tasks for developers. This may not be the case in practice. The threat is however mitigated given that, in any case, we have shown that the test cases are often unavailable when the bug is reported.

5.16 Related Work

Fault Localisation. A recent study [131] stated fault localisation is a critical task affecting the effectiveness of automated program repair. Several techniques have been proposed [184, 231, 249] and they use different information such as spectrum [5], text [242], slice [147], and statistics [122]. The first two types of techniques are widely studied in the community. SBFL techniques [4, 77] are widely adopted in APR pipelines since they identify bug positions at the statement level. However, they have limitations on localising buggy locations since it highly relies on the test suite [131]. IRFL [120] leverages textual information in a bug report. It is mainly used to help developers narrow down suspected buggy files in the absence of relevant test cases. For the purpose of our study, we have proposed an algorithm for localising the faulty code entities at the statement level.

Patch Generation. Patch generation is another key process of APR pipeline, which is, in other words, a task searching for another shape of a program (i.e., a patch) in the space of all possible programs [115,139]. To improve repair performance, many APR systems have been explored to address the search space problem by using different information and approaches: stochastic mutation [117,238], synthesis [138, 255, 257], pattern [48, 72, 76, 88, 110, 114, 132, 135, 137, 200], contract [35, 235], symbolic execution [172], learning [16, 63, 140, 195, 213, 243], and donor code searching [84, 155]. In iFixR, patch generation is implemented with fix patterns presented in the literature since it may make the generated patches more robust [204].

Patch Validation. The ultimate goal of APR systems is to automatically generate a *correct* patch that can actually resolve the program defects rather than satisfying minimal functional constraints. In the beginning, patch correctness is evaluated by passing all test cases [88, 114, 238]. However, these patches could be overfitting [112, 190] and even worse than the bug [210]. Since then, APR systems are evaluated with the precision of generating correct patches [76, 132, 241, 255]. Recently, researchers explore automated frameworks that can identify patch correctness for APR systems automatically [113, 254]. Our approach validates generated patches with regression test suites since fail-inducing test cases are readily available for most of the bugs as described in Section 5.10.

5.17 Summary

In this study, we have investigated the feasibility of automating patch generation from bug reports. To that end, we implemented iFixR, an APR pipeline variant adapted to the constraints of test cases unavailability when users report bugs. The proposed system revisits the fundamental steps, notably fault localisation, patch generation and patch validation, which are all tightly-dependent to the *positive test cases* [238] in a test-based APR system.

Without making any assumptions on the availability of test cases, we demonstrate, after re-organising the Defects4J benchmark, that iFixR can generate and recommend priority correct (and more plausible) patches for a diverse set of user-reported bugs. The repair performance of iFixR is even found to be comparable to that of the majority of test-based APR systems on the Defects4J dataset. We evaluate iFixR on the Defects4J dataset, which we enriched (i.e., faults are linked to bug reports) and carefully-reorganised (i.e., the timeline of test-cases is naturally split). `iFixR` generates genuine/plausible patches for 21/44 Defects4J faults with its IR-based fault localiser. `iFixR` accurately places a genuine/plausible patch among its top-5 recommendation for 8/13 of these faults (without using future test cases in generation-and-validation).

6 Exploring Generic Concepts of Patching

Template-based program repair research is in dire need of common to express fix patterns in a standard and reusable manner. We propose to build on the concept of *generic patch* (also known as semantic patch), which is widely used in the Linux community to automate code evolution. We advocate that generic patches could provide at the same time a unified representation and a specification for fix patterns. Generic patches are indeed formally defined, and there exists a robust, industry-adapted, and extensible engine that processes generic patches to perform control-flow code matching and automatically generates concretes patches based on the specified change operations.

In this work, we present the design and implementation of a repair framework, FlexiRepair, that explores generic patches as the core concept. In particular, we show how concretely generic patches can be inferred and applied in a pipeline of Automated Program Repair (APR). With FlexiRepair, we address an urgent challenge in the template-based APR community to separate implementation details from actual scientific contributions by providing an open, transparent and flexible repair pipeline on top of which all advancements in terms of efficiency, efficacy and usability can be measured and assessed rigorously. Furthermore, because the underlying tools and concepts have already been accepted by a wide practitioner community, we expect FlexiRepair's adoption by industry to be facilitated. Preliminary experiments with a prototype FlexiRepair on the IntroClass and CodeFlaws benchmarks suggest that it already constitutes a solid baseline with comparable performance to some of state of the art.

Contents

6.1 Overview

In the race for achieving the old software engineering dream of automating program repair, approaches that leverage fix patterns currently have the lead (in terms of how many benchmark bugs can be fixed) [134]. Unfortunately, despite the excitement of this momentum in the research community, practitioners expectations are not met, and full integration in industrial settings remain anecdotal. Initial experimental attempts to large-scale application of automatic bug fixing suggest however that pattern-based patch generation fits the current practice of software engineering: ❶ At Facebook, Getafix [12] and SapFix [148] suggest fixes for in-house software by learning patterns using "hierarchical clustering to many thousands of past code changes that human engineers made, looking at both the change itself and also the context around the code change". ❷ In the Linux open-source ecosystem, the Coccinelle [177] code transformation engine, which builds on pattern-like specifications written by developers, has been leveraged to automatically generate over 6 000 patches [105] that were accepted in the kernel code base. ❸ Besides repair, Ubisoft designed Clever [168] to detect risky commits at commit-time using patterns of programming mistakes from the code history.

Recently, Liu et al. [133] proposed to revisit the performance of automated program repair (APR), carefully searching to identify the pattern databases that were available in the literature. Their experience report suggests that researchers do not actually share a common definition of what constitutes a repair pattern: levels of abstraction vary significantly and their immediate exploitation is often impossible as a transferable ingredient. Koyuncu et al. [99] and Ueda et al. [229], in two independent works, pointed out that fix patterns should be made *tractable* (i.e., they should be a clearly identifiable artefact in the repair pipeline towards explaining the patch generation decisions) and *editable* (i.e., APR users should be enabled to intervene to correct these patterns manually to take into account project specificities). On top of these concerns, the full APR pipeline generally suffers from a lack of:

- **Practicality**: A large body of the literature in APR present approaches that target well curated benchmarks with several constraints (e.g., test cases are readily available for the identified bugs) which may not be the case in practice. Although recent works [100] have started to investigate the use of bug reports, their pipelines remain heavily driven by test suites (e.g., for validation).
- **Flexibility**: Regardless of the patch generation process (i.e., heuristic-based, constraint-based, or template-based following the taxonomy proposed in [118]), the available change transformations are generally limited to small mutations operators which are tightly embedded in the proposed algorithms. Seldom, an approach allows third party members in the community to readily edit and extend the list of possible code transformations.
- **Transparency**: The repair approaches suggest patch candidates from a search space. However, in most of the case, the origins of the patch candidates, i.e., how they are discovered, is missing. This intractability remains a big obstacle for transparency.

Overall, we note that on the road to automated program repair, the practitioner community is looking for techniques that can rapidly recommend patches that may be manually validated by developers. Indeed, these appear to be acceptable in various industrial settings so far. It may thus be worthwhile to drain some research effort into building an automatic patch generation system that is based on robust, agile and tractable techniques for inferring code transformation strategies.

We present the core concept behind FlexiRepair, a flexible, transparent and practical APR pipeline. We have initiated this FlexiRepair and call for the community to commit on working on its building blocks for delivering reliable tool support for practitioners in the context of program repair. FlexiRepair is built on top of well-accepted software maintenance concepts in the Linux community, notably the concept of *generic patch* (more known as *semantic patch* in recall for the specification language: the Semantic Patch Language [28]). **We use *generic patch* specification as the tractable notation for fix patterns**. The main contributions of our work are as follows:

- We propose a critical review of template-based APR steps and suggest the design of a patch generation system around the concept of *generic patch* whose underlying definition and structure is borrowed from the Linux community toolbox.
- We initialise an open framework for program repair. The proposed pipeline, FlexiRepair, transparently uses state of the art building blocks that can be customized.
- We evaluate the prototype pipeline, using available building blocks from the literature. Performance is measured with *C* program repair benchmarks.

6.2 Related Work

Program repair at a glance. Patch generation is one of the key tasks in software maintenance since it is time-consuming and tedious. If this task is automated, the cost and time of developers for maintenance will inevitably be reduced dramatically. To address this issue, many automated techniques have been proposed for program repair [118, 163]. Ultimately, program repair is about traversing a search space of patch candidates that are generated by applying change operators to the buggy program code. Depending on how a technique conducts the search and constructs the patches, it can be considered as *heuristics-based* [76, 88, 133, 238] or *constraint-based* [156, 172, 255] following the taxonomy proposed by Le Goues *et al.* [118]. If such a technique further leverages learning mechanisms to infer transformation patterns or to build patch models or even to predict patches, it is considered as *learning-aided* [63, 137, 140].

In the last decade, most proposed techniques in the literature present repair pipelines where patch candidates are generated then validated against a program specification, generally a (weak) test suite. We refer to them as *generate-and-validate* test-suite based repair approaches and focus FlexiRepair framework under this practical repair scheme. The genetic programming-based approach proposed by Weimer *et al.* [238], as well as follow-up works, appeared only valid for hypothetical use cases. Nevertheless, in the last couple of years, two independent reports have illustrated the use of literature techniques in actual development flows: in the open source community, the Repairnator project [230] has successfully demonstrated that automated repair engines can be reliable: open source maintainers accepted and merged patches which were suggested by an APR bot. At the premises of Facebook, the SapFix repair system has been reported to be part of the continuous integration pipeline [148] while Getafix was used there at a large scale [12].

Given fault localisation information that pinpoints the code locations in the program that are the most likely to be buggy, test suite program repair approaches apply syntactic transformations to generate patches. Early techniques such as GenProg [117, 238] relied on simple mutation operators to drive the genetic evolution of the code. More widespread today are approaches that build on fix patterns [88] (also referred to as fix templates [135] or program transformation schemas [72]) learned from existing patches. Several APR systems [48, 72, 88, 99, 129, 131, 132, 132, 133, 135, 153, 200] implement this strategy by using diverse sets of fix patterns obtained either via manual generation or automatic mining of bug fix datasets. Unfortunately, whether they are generated on-the-fly (e.g.. with SimFix [76] and CapGen [241]) or stored in a database, fix patterns remain an elusive concept.

☞ In this work, our aim is to establish generic patches specified via the Semantic patch language as the formal notation for abstracting and defining fix patterns.

Fix patterns inference for program repair. While the literature includes a large body of work on change patterns [54, 55, 151], and more generally on change redundancies [91, 93, 130, 161, 173, 181, 272], very few approaches have actually leveraged their "discovered" patterns again to instantiate repair patches.

Nevertheless, specific bug patterns have been mined to build fixing engines: Livshits and Zimmermann [136] discovered application-specific repair templates by using association rule mining on two

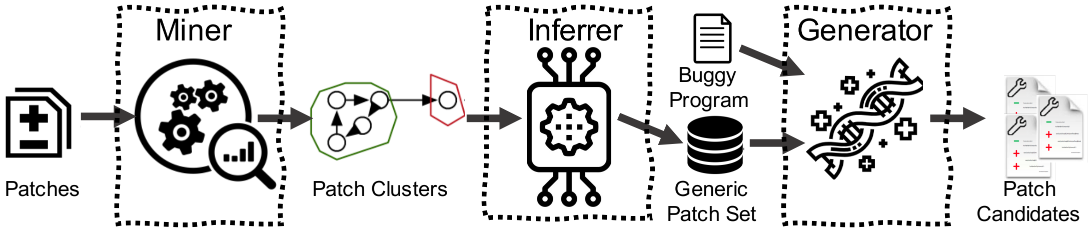


Figure 6.1: The FlexiRepair pipeline.

Java projects while Hanam et al. [64] have developed the BugAID technique for discovering most prevalent repair templates in JavaScript.

DevReplay [229] is a recent static analysis tool that suggests source code changes based on a project's git history. The proposed changes can be edited by users without requiring knowledge about the AST.

FixMiner [99] is an automated approach to mining relevant and actionable fix patterns based on an iterative clustering strategy applied to atomic changes within patches. The goal of FixMiner is to infer separate and reusable fix patterns that can be leveraged in other patch generation systems. This approach provides an appealing building block in the context of the FlexiRepair framework. Unfortunately, its patterns are also not immediately actionable; they must be manually integrated into a repair engine, which requires tedious and error-prone hard-coding of bug-fixing patterns. Additionally, FixMiner patterns do not contain any code token information: they have holes. The donor code should be searched before generating a concrete patch, which may lead to various non-sensical patches. FixMiner currently supports only Java, and it does not provide any end-to-end traceability (i.e. we do not know from where the pattern has been inferred).

☞ We borrow some ideas from the FixMiner approach for computing patch similarity towards inferring patterns. In particular, we find their rich AST edit script to be appealing for building the prototype implementation of FlexiRepair.

Generic patches in the literature. There has been some work addressing the problem of considering a set of patches and attempting to find a "generic patch" that summarises the change that is common across the patches. Chawathe et al. proposed a seminal method to detect changes to structured information based on an ordered tree and its updated version [33]. The goal was to derive a compact description of the changes with the notion of minimum cost edit script which has been used in the recent ChangeDistiller and GumTree tools. Spdiff [9, 10] was then a promising approach that considered inferred change patterns from a set of patches. It was however found to scale poorly to a large number of patches, and to have constraints in producing ready-to-use patterns that can be used (e.g., by the Coccinelle matching and transformation engine [28]). Recently, Serrano et al. [206] proposed Spinfer as a tool-supported approach to ease large-scale changes across many source files in Linux by suggesting transformation rules to developers, inferred automatically from a collection of examples.

☞ Spinfer builds on the notation of "generic patch" (also referred to as "semantic patch"), which Linux developers are already familiar with, thanks to the wide adoption of the Coccinelle [177] transformation engine and the associated Semantic Patch Language. We will rely on this building block for the inference of fix patterns in the prototype version of FlexiRepair.

6.3 The FlexiRepair Framework

FlexiRepair builds on the momentum of template-based program repair, which has been shown successfully in fixing a variety of bugs in APR benchmarks. To date, these approaches are among the most effective (in terms of the number of benchmark bugs that are fixed) repair tools in the literature. Relevant approaches in the literature (e.g., TBar [133], AVATAR [132], CapGen [241], SimFix [76]) are often provided in monolithic tooling which prevents extension, adaptation and even application on real-world code bases beyond those targeted by initial experimental validations.

> *In this work, we propose to initiate a community-wide effort to build a flexible, transparent and practical framework for template-based program repair to (1) enable better assessment of research advancements, and (2) facilitate the adoption of APR by software maintainers.*

FlexiRepair is carefully designed to ensuring that its users have control over important steps of the patch generation process. In particular, we consider the following critical questions:

❶ **Where should we mine repair transformations?** Template-based program repair systems, whether they leverage specifically pre-defined mutation operators, infer code transformations on-the-fly or rely on offline-inferred fix patterns, they generally build on data of existing code bases (preferably with a large history of code changes). If the source of mining is not appropriate (e.g., limited recurrent changes or changes associated with domain-specific bugs), the mined patterns may be irrelevant for the program that is targeted for repair.

❷ **How are fix patterns inferred?** A challenge that has been highlighted in two recent independent studies [99, 229] is that fix patterns discussed in the APR community are largely intractable artefacts. If the underlying fix patterns cannot be manipulated (i.e., checked and edited) by practitioners, the adoption of the integrating APR tool will be largely hindered.

❸ **How are patches generated?** Besides fault localisation information which generally drives the selection of fix patterns, the application of code transformations generally follows various ad-hoc recipes and involve empirical design choices for fix pattern matching and donor code search. If these activities cannot be ensured to be deterministic, industry adoption of APR cannot be ensured.

6.3.1 Execution steps of FlexiRepair

We propose to build an APR pipeline that addresses the issues raised in the aforementioned questions. Figure 6.1 illustrates an overview of the FlexiRepair.

The pipeline takes the code repositories that the maintainer judge to be relevant for learning code transformations as its input. This set of code repositories can be constituted by the single source code repository associated with the program under repair. Then, each of the questions formulated above is addressed by a major component involved in a specific step of the FlexiRepair pipeline :

- Miner analyses the structural similarity between input repository patches and yields clusters that can be tuned by FlexiRepair users to take into account the recurrence level of code transformations that will be supported by the patch generation.
- Inferrer then abstracts fix patterns from each retained cluster and specifies it in a format that can be inspected (for relevance) and edited (to account for specific maintenance style requirements).
- Generator finally builds the concrete patches for the given buggy programs, after attempting to match fix patterns to the appropriate code locations (i.e., the likely buggy code locations).

Instead of re-inventing new algorithms and prototyping tools that would require extensive vetting before adoption, we propose to bootstrap the FlexiRepair pipeline by relying on tried-and-true technologies that software maintenance is already familiar with. Concretely, we have identified a code transformation tool that is part of the Linux kernel developer toolbox since 2008 and which is now increasingly used to automate large-scale changes in kernel code. This tool, Coccinelle [177] builds on a concept of *semantic patch* that allows developers to write transformation rules using a diff-like syntax. **In this work, we will use the term "*generic patch*" instead to refer to the specification of transformation rules that can be given as input to Coccinelle.** A generic patch is thus an abstraction that uses metavariables to represent common but unspecified subterms (e.g., any variable) and notation for reasoning about control flow paths.

Given the standing of Linux development practices in the software development community, the adoption of a tool such as Coccinelle, and its underlying concepts, is a strong signal that it fits with industry standards. We therefore propose to build the FlexiRepair pipeline on top of the Coccinelle engine.

> *A fix pattern in* FlexiRepair *is a* `generic patch` *that is specified using the specification language of Coccinelle, which is now integrated to the Linux development toolbox.*

6.3.2 Overview of the SmPL Language

The Coccinelle tool is an example of a public research effort that gained traction in industrial settings, thanks to support from the open source community. It was initially designed to document and automate *collateral evolutions* [177] in the Linux kernel source code, but is now used in a variety of other code bases as a base engine for performing control-flow-based program searches and transformations in C code [105]. Coccinelle integrates a static analysis that is specified using control-flow sensitive concrete syntax matching rules. Search (i.e., identifying code fragments that match a pattern) and transformations (i.e., generating patches following the fix pattern) are specified via the Semantic Patch language (SmPL), and executed by a dedicated transformation engine. Although the Linux community refers to the SmPL specifications as "semantic patches", we will refer to them in FlexiRepair as "generic patches" to reflect the idea that they are abstract patterns that must be "concretised" into generated patches.

Although SmPL specifications can contain OCaml or Python code, allowing to perform arbitrary computations, in this work we focus on its pattern matching and code transformation capabilities. Listing 6.1 provides an example of generic patch (as an SmPL specification). The patch goal is :

- (1) to identify all code locations where there is an attempt to access a field of struct whose pointer has not been safely checked beforehand in the control flow. Indeed, if the pointer (*param*) is NULL, the dereference would lead to a segmentation fault (and a crash in the case of an operating system code).
- (2) to produce a corrective patch (i.e., adding check and early return statement) at all places where such an unsafe dereference can take place.

The generic patch is constituted of a single rule named "unsafe_dereference" which defines five metavariables (lines 2-4): T (type) which represents any data type; p, which represents an arbitrary position in the source program; fn (function name), *param* (parameter name) and fld (data structure field name), which represent arbitrary identifiers. *Metavariables* are bounded by matching the code pattern against the source code. For example, the pattern fragment on line 6 (`fn(..., T *param, ...)`) will cause fn to be bounded to the name of a function in its definition, and cause *param* to be bounded to any pointer parameter name. The notation @p binds the position metavariable p to information about the position of the match of the preceding token. Once bounded, **a metavariable must maintain the same value within the current control-flow path**; thus, for example, the occurrences of param on lines 6-13 must all match the same expression. The fix pattern (lines 6-15) therefore consists of essentially C code, mixed with a few operators to raise the level of abstraction so that a single generic patch can apply to many code sites.

```
1 @unsafe_dereference exists@
2 type T;
3 position p;
4 identifier fn, param, fld;
```

```
5  @@
6  fn(.., T *param, ...){
7  ... when != param = new_val
8      when != param == NULL
9      when != param != NULL
10     when != IS_ERR(param)
11 +   if (param == NULL)
12 +     return
13     param−>fld@p
14 ... when any
15 }
```

Listing 6.1: Example of generic patch

6.3.2.1 Sequences abstraction

The main abstraction operator provided by SmPL is '...', representing a sequence of terms. In line 6, '...' represents the remaining parameters of a function that appear before and after a given parameter is matched in the parameter list; in line 7, '...' represents the sequence of statements reachable from the begin of the definition of a function along any control-flow path. By default[1], such a sequence is quantified over all paths (e.g., over all branches of a conditional block), but the annotation "`exists`", next to the rule name, indicates that for this "unsafe_dereference" rule, the matching should be done even for one path. It is also possible to restrict the kinds of sequences that '...' can match using the keyword `when`. Lines 9-12 use `when` to indicate that there should be no reassignment of param nor any check on the validity of the param pointer value before reaching the dereference that consists in accessing a field fld in the corresponding data structure.

An SmPL rule only applies when it matches the code completely. Consider the example of buggy code in Listing 6.2. The rule `unsafe_dereference` matches the parameter of type struct person * on line 1 and the dereference on line 6 as it exists a control-flow path where the validity of *pers* is not checked. The metavariable *fn* (cf. Listing 6.1)is bound to the identifier *get_age*, and the metavariable *param* is bound to *pers*. The metavariable *p* is bound to various information about the position of the dereference, such as the file name, line number, character number (on the line).

```
1 int get_age(int alive, struct person *pers, char *context){
2 int age=0;
3 if (alive == 1 && pers !=NULL)
4     age=pers−>age_death − pers−>age;
5 else
6     age = pers->age;
7 return age;
8 }
```

Listing 6.2: Example of buggy code with an unsafe dereference

6.3.2.2 Disjunctions and Nests

Besides the '...', SmPL provides **disjunctions**, (pat_1 | ... | pat_n), and **nests**, < ... *pat* ... >. A nest < ... *pat* ... > in SmPL matches a sequence of terms, like '...'. However, additionally, it can match zero or more occurrences of *pat* within the matched sequence. Another form of nest exists for matching one or more occurrences of pat. By analogy to the + operator of regular expressions, this form is denoted < +... *pat* ...+ >.

[1] This default behaviour can also be explicitly stated using the "`forall`" annotation

The examples discussed above illustrate the abstraction power that generic patches provide in the activity of propagating fixes. In the next section, we present the steps for:

- **Regrouping patches** into clusters where bug fix code transformations are made with similar patterns.
- **Automatically generating generic patches** from clusters of patches in order to populate the repair template databases.
- **Performing patch generation** in practice given an identified buggy code location (even at a coarse granularity)

6.3.3 Patch clustering

The goal of the Miner is to perform patch clustering, i.e., to group together the code changes that are representing a repeating code context and change operations. In order to convey the full syntactic and semantic meaning of the code change and to discover clusters of patches that are sharing a common representation, we leverage the rich AST edit script representation proposed by Koyuncu et al. [99].

⟨*richASTEditScript*⟩ → ⟨*node*⟩+

⟨*node*⟩ → '---'* ⟨*move*⟩ | '---'* ⟨*delete*⟩ | '---'* ⟨*insert*⟩ | '---'* ⟨*update*⟩

⟨*move*⟩ → 'MOV' ⟨*astNodeType*⟩ @@ ⟨*tokens*⟩ '@TO@' ⟨*astNodeType*⟩ '@@' ⟨*tokens*⟩ '@AT@'

⟨*delete*⟩ → 'DEL' ⟨*astNodeType*⟩ '@@' ⟨*tokens*⟩ '@AT@'

⟨*insert*⟩ → 'INS' ⟨*astNodeType*⟩ '@@' ⟨*tokens*⟩ '@TO@' ⟨*astNodeType*⟩ '@@' ⟨*tokens*⟩ '@AT@'

⟨*update*⟩ → 'UPD' ⟨*astNodeType*⟩ '@@' ⟨*tokens*⟩ '@TO@' ⟨*tokens*⟩ '@AT@'

Grammar 6.1: Notation of rich AST edit script

A rich AST edit script, whose grammar is illustrated in Grammar 6.1, encodes the information about the AST node types in a change diff-tree, the repair actions performed, the raw tokens involved as well as the parent-child relationship among the nodes. We consider only code context and change operation presentation (cf. Figure 6.2) to detect similar changes and group them into clusters of similar code changes. The objective of this step is to ensure that we can reduce the noise in pattern inference, regrouping together patches that perform similar changes actions, and potentially filtering out cases where the redundancies of changes are limited.

```
UPD expr_stmt
——— UPD expr
—————— UPD call
————————— UPD name
————————— UPD argument_list
———————————— DEL argument
——————————————— DEL expr
—————————————————— DEL literal:string
```

Figure 6.2: An example code context and change operation presentation.

6.3.4 Generic Patch Inference

The goal of the Inferrer is to derive generic patches from the clusters of similar concrete patches that have been mined in the previous step. We build on a recent work by Serrano et al. [206], which showed that it is possible to generate SmPL transformation rules by learning from examples automatically. The approach considers both similarities among code fragments and among control flows associated with the changes to identify change patterns and specify transformation rules.

In practice, the idea is to abstract over common changes across the examples, incrementally extending a pattern until obtaining a rule that describes the complete change, respecting both control-flow and data-flow relationships between the fragments of the code. To that end, each patch in a cluster is used to reconstitute the before- and after-change files, then the Inferrer identifies sets of common removed or added terms across the examples, and further generalises these terms in each set into a pattern that matches all of the terms in the set, and finally integrates these patterns into transformation rules that respect both control-flow and data constraints exhibited by the examples.

Figure 6.3 illustrates an example of the inferred generic pattern. It encodes the information of how to transform code, the locations of where the pattern is inferred, as well as statistics on the recall (i.e., the percentage of expected changes in the examples that are obtained by applying the inferred generic patch), precision (i.e., the percentage of changes obtained by applying the inferred generic patch that is identical to the expected changes in the examples). We benefit in FlexiRepair from the SmPL-specified generic patch notation which is close to C: it makes the patterns understandable to the user and even allows the user to improve the script or adapt it to other uses, hence contributing to the **transparent** schema of program repair where patterns are tractable.

```
@@
identifier I0;
@@
- char I0;
+ int I0;
// Infered from: (MultiMarkdown-6/{prevFiles/prev_626381_73d644_Sources#libMultiMarkdown#aho-corasick.c,revFiles/626381_73d644_Sources#libMultiMarkdown#aho-corasick.c}: ac_trie_search)
// Recall: 0.11, Precision: 1.00, Matching recall: 0.50
```

Figure 6.3: An example of generic patch.

6.3.5 Code Transformation with Generic Patches

Given that we leverage the SmPL language to specify generic patches, code transformation is provided for free by the Coccinelle search and transformation engine. The engine takes as input a generic patch and a source code file that it parses. Then it performs a control-flow matching to identify code locations whose shape fit with the structure of the code structure targeted by the generic patch. Taking the metavariables values bounded at each matched code location, it then generates the necessary concrete patches. This engine thus provides two essential advantages over existing generate-and-validate repair pipelines:

- (1) there is no need for a fine-grained bug localisation engine. A coarse-grained localiser that points to buggy files can be leveraged. Thus, even IR-based bug localisation tools which produce results at file level without requiring test cases can be relevant (as advocated by recent work [100]).
- (2) the search for donor code is facilitated by the use of metavariables in the generic patch, allowing for the transformation engine to infer and track tokens across the control-flow, and thus maximising the chances of producing sensical patches (i.e., patches that at least make the program compile).

6.4 Study Design

We now overview some details of our experimental validation. The objective in this study is not to build a novel state of the art repair tool, but to rather offer a new perspective into a framework for template-based program repair with at its core the concept of generic patch for specifying fix patterns (aka fix templates). Before presenting the results, we discuss the dataset of code repositories that we build for mining similar patches and further inferring generic patches. Then, we present the benchmarks used to assess the overall performance of FlexiRepair as our prototype implementation. Finally, we overview the implementation choices made for both the patch clustering and pattern inference steps.

6.4.1 Subjects

FlexiRepair provides a flexible interface to its user, who can decide either to use a specific code repository to mine the fix patterns, or to use the pre-constructed fix pattern database shipped with the framework. To build this database, we needed first to collect a large set of the code repositories with a long history of code changes.

The subjects that are included in our dataset are collected: i) by manual identification of popular *C* repositories from Github, Gitlab, Savannah with a large code history and ii) systematically by leveraging the build activities in Travis CI. For the latter, we refer to the data of Durieux et al. [47], which contains all the Travis CI jobs executed between 30 September 2018 and 22 January 2019 by 272 917 projects. From their dataset, we identified 2 858 *C* repositories. We further curate this dataset based on the repository properties from Github (i.e., commit count, watchers count, forks count etc.). Eventually, we select the repositories i) that are not forks, ii) having at least 200 commits, iii) at least 10 watchers iv) and at least 10 forks. Our dataset eventually includes 351 repositories. Table 6.1 lists some of the major ones.

Table 6.1: Some of selected repositories used in our study.

Systematic identification			
repository	commitCount	watchers	forks
xqemu/xqemu	68836	462	49
git/git	59910	33398	19513
greenplum-db/gpdb	56052	4073	1171
MonetDB/MonetDBLite-C	54946	26	13
panda-re/panda	54869	1640	393
Manual identification			
repository	commitCount	source	
linux	949406	git.kernel.org	
freebsd	271000	github.com	
FFmpeg	98901	github.com	
cmake	49611	gitlab.kitware	
gtk	65679	gitlab.gnome	

6.4.2 Assessment Benchmarks

We selected the IntroClass [116] and CodeFlaws [220] datasets to empirically assess FlexiRepair.

The IntroClass dataset is a benchmark of small *C* programs collected from classroom assignments of students. It includes 998 defects, 778 of them being associated with an instructor constructed black-box test suite and 845 is associated with a white-box test suite created using KLEE [30] (a symbolic execution tool that automatically generates tests).

CodeFlaws benchmark consists of 3 902 defects collected from C programs developed during programming contests. The benchmark is associated with two sets of test-suites: i) a test suite given to repair tools for generating repair ii) a held-out test suite for validating the correctness of patches.

Table 6.2 lists some statistics about the benchmarks.

Table 6.2: Basic statistics of Benchmarks.

Benchmark	# of Defects	Size of Test Suite I	Size of Test Suite II	LOCs
Codeflaws	3902	2-8	5-350	1-322
Introclass	998	6-9	6-10	13-24

6.4.3 Implementation Choices

FlexiRepair aims for flexibility and extensibility such that practitioners may tune parameters and adapt the framework to their requirements. We recall that we have made the following parameter choices in the FlexiRepair:

- Repository selection is made based on the C programs with a large commit history and which are actively used.
- Change size in a patch is limited to have at most 50 changed lines.
- Patch spread is limited such that each patch contains at most 3 hunks.
- Timeout for generic patch inference is set to 900 seconds for each patch cluster.

6.5 Assessment

We assess the prototype framework of FlexiRepair via performing experiments that answer the following research questions.

6.5.1 Research Questions

RQ-1: *To what extent can the application of* Miner *and* Inferrer *produce generic patches from the collected code repositories?*

RQ-2: *Where did* FlexiRepair *find relevant redundant changes to mine the generic patches?*

RQ-3: *What is the repairability performance of* FlexiRepair*?*

RQ-4: *What is the efficiency performance of* FlexiRepair*?*

6.6 Results

6.6.1 Generic Patch Inference Capability

We first assess the relevance of the patch clusters yielded by the Miner component of FlexiRepair. Then, we look at the generic patches yielded by the Inferrer implementation.

Miner Assessment. Performance of Miner is evaluated through the clusters that it yields. The objective is to estimate whether it can find enough cases of recurrent changes within patches collected from project repositories to form clusters. A given patch cluster will contain all the patches having a similar code change hunk. Table 6.3 overviews the statistics of clusters yielded by Miner by taking as input the dataset of repositories presented in Section 6.4. The implementation choices presented in

Section 6.4 are also followed. Overall, 350 676 code hunks have been extracted. Among these hunks, we noticed that 110 949 (∼32%) are unique code hunk, and thus, they cannot be part of a cluster. For the remaining 239 727 code hunks (∼68%), there exists at least one other code hunk, among the 350 676 code hunks, which is identical. They can thus form clusters of more than one patch. Overall, among these 239 727 code hunks, we identified 31 310 patch clusters (i.e., the code hunks of each patch of a cluster are identical).

Table 6.3: Statistics on Patch Clusters .

Total # of hunks	# unique hunks	# hunks which can form a cluster of at least 2 patches	# clusters
350 676	110 949	239 727	31 310

Figure 6.4 shows the size distribution of the patch clusters. A majority of clusters, i.e., 16 081 (∼51%), are formed by two recurrent code change hunks only. Conversely, 2394 (249+1 023+875+247, ∼7.6%) clusters contain at least 10 recurrent code change hunks.

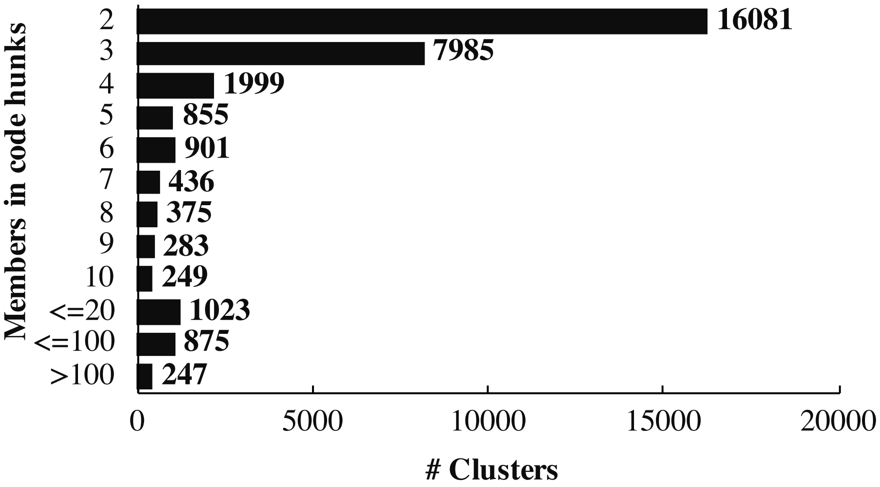


Figure 6.4: Distribution of the patch cluster sizes.

We further investigate how the cluster elements are spread across patches. To that end, we follow the categorisation proposed by Koyuncu et al. [99].

- A *vertical cluster* is a cluster whose code change hunk is recurrent within a single patch. Such clusters are generally formed when we have patches that developers commit to performing a single type of change (e.g., change *kmalloc* call to *kzalloc* calls) across several code locations.
- An *horizontal cluster* is a cluster whose code change hunk is recurrent across several patches. Such clusters are formed when a code change (e.g., add a missing NULL check) is implemented by different developers for different code locations.

Table 6.4 overviews the statistics of clusters yielded. Most of the clusters (24 230) are horizontal clusters. This suggests that the same code changes are often spread among different patches, any or all of which may be used to infer the common generic patch. The vertical clusters can also be useful for inferring generic patches: they represent large patches making the same changes at once at several locations (e.g., collateral evolutions in Linux are applied through vertical patches [177]).

> **RQ-1.1**: Miner *is practical. It is able to identify patch clusters (i.e., recurrent patch sets) of various sizes.*

Table 6.4: Statistics on Patch Clusters Spread

Vertical			Horizontal		
# Clusters	# Patch	# Hunk	# Clusters	# Patch	# Hunk
3178	3178	7565	24230	75691	75691

* A generic patch can simultaneously be vertical (when it is associated to several changes in hunks of the same patch) and horizontal (when it appears as well within other patches).

Inferrer Assessment. We assess the ability of Inferrer to analyses changes within patch clusters and derive a generic patch (i.e., abstract the relevant fix pattern and specify it with the SmPL notation). Our implementation uses Spinfer as a backend for matching control-flow similarities. Preliminary experiments revealed that the approach is sensitive to the noise among patches. We expect our Miner step to have provided homogeneous patch clusters.

Table 6.5 overviews some statistics on the inferred generic patches. From the 31 310 patch clusters obtained with Miner, Inferrer was able to successfully yield a generic patch for 20 467 (∼65%) clusters. The remaining clusters (∼35%) do not lead to any generic patch either because of the timeout value of 900 seconds set for analysing each patch cluster, or because they do not exhibit the necessary data or control-flow dependencies to satisfy any inference. Note that the initial generic patch inferred from a given cluster can contain several rules. We consider each transformation rule as a generic patch on its own. Eventually, we are left with 68 368 atomic generic patches (i.e., generic patches with a single transformation rule).

Note that in the middle column of Table 6.5, we also report the number of code hunks that have been used to infer the generic patches. Overall, 125 483 (∼52%) out of 239 727 code hunks contributed to a generic patch.

Table 6.5: Inferred Generic Patch statistics.

# Patch Clusters	# Code hunks	# "Atomic" Generic Patches
20 467	125 483	68 368

Table 6.6 lists five of the most frequently observed generic patches in our dataset. We further manually investigate these generic patches by checking the corresponding commits in the repositories in order to understand the nature of the changes described by the developers.

We discover that two generic patches (generic patches #1 and #3 in Table 6.6) have been generated from patches that were actually automatically generated to automate some evolutions at large scale across Linux: the relevant commit logs even mention the Coccinelle tool being used.

The second generic patch (id `expr_stmt_4_32`), is spread among 14 projects (as we can see in the Frequency column of Table 6.6) and the associated commits are often described with "Fix coding style" (indeed, the generic patch simply removes brackets). The generic patch `block_content_18_3` is inferred from 3 different projects. This generic patch fixes a memory mapping issue. Finally, the generic patch `if_stmt_8_8` switches the order of the expressions in the condition of the `if statement`, to alter the control flow. A corresponding commit log summarises this behaviour as *"Put CONFIG* first in if(). This may fix build failures with EAC3 disabled and is more consistent"*.

To conclude this RQ, we check from which repositories the generic patches have been inferred. Table 6.7 lists the Top-10 projects, which contributed to the pattern inference. We note that all of these projects have large code histories. Overall, from the 351 repositories used to mine the clusters and infer the generic patterns, 301 contributed to pattern inference. It is possible that the remaining 50 projects do not contribute because of the filtering constraints imposed in our implementation

Table 6.6: Frequently observed generic patches.

Frequency		generic patch
Hunk Function File Patch Project	202 99 99 116 1	@block_content_42_0@ identifier I4, I0; expression E1, E2, E3; @@ - struct resource *I0; ... - I0 = platform_get_resource(E1, IORESOURCE_MEM, E2); - E3->I4 = devm_ioremap_resource(&E1->dev, I0); + E3->I4 = devm_platform_ioremap_resource(E1, E2);
Hunk Function File Patch Project	178 149 83 48 14	@expr_stmt_4_32@ expression E0; @@ - return (E0); + return E0;
Hunk Function File Patch Project	100 50 32 4 1	@block_content_12_25@ expression E0; @@ - free(E0); - E0 = NULL; + FREE_AND_NULL(E0);
Hunk Function File Patch Project	84 21 19 24 3	@block_content_18_3@ expression E2, E1, E0; @@ - memory_region_init_ram(E0, NULL, E1, E2, &error_abort); - vmstate_register_ram_global(E0); + memory_region_allocate_system_memory(E0, NULL, E1, E2);

Hunk
Function
File
Patch
Project
	78
	37
	66
	7
	3
		@if_stmt_8_8@
		binary operator B1 = {== ,&& };
		expression E0, E2;
		@@
		- if (E0 B1 E2)
		+ if (E2 B1 E0)
		{
		...
		}

choices, or simply because they do not contain enough recurrent code change context from we generic patches can be inferred.

Table 6.7: Top-10 projects contributed to pattern inference.

projects	freebsd	linux	qemu	wireshark	FFmpeg	php-src	xqemu	vlc	panda	gtk
occurrences	11812	11419	9997	9337	7187	7090	6288	5249	4740	4325

> **RQ-1.2**: Inferrer *successfully yields generic patches for a large number of clusters: some generic patches are summarized patterns of changes that spread across several projects.*

6.6.2 Generic Patch tractability

We investigate potential relationships between the distributions of code change locations and the performance of pattern inference, in order to estimate the adequate locations for optimising the search of generic patches.

Generic patches are inferred from hunks in a cluster. Note that, in a cluster, when the code context and change operation of several hunks are syntactically identical, we consider them as a single same hunk in this research question. Figure 6.5 shows the distribution of the generic patches in terms of the number of hunks that were used to infer them. Overall, from the 68 368 inferred generic patches, 40 529 (∼60%) is inferred from a single hunk.

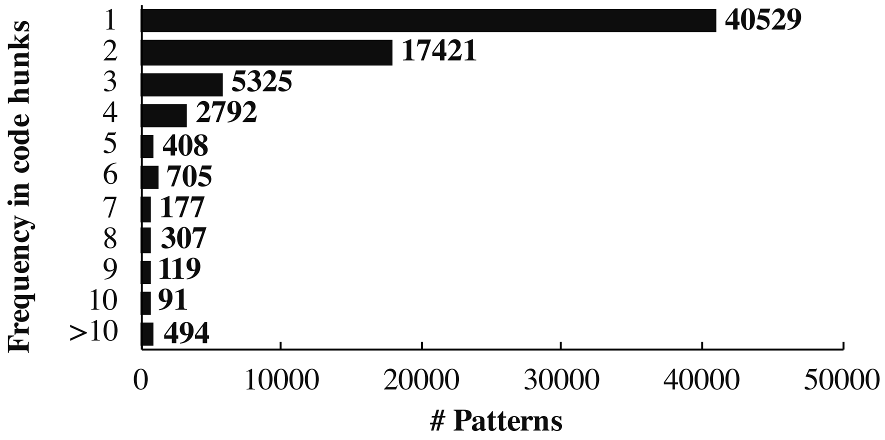


Figure 6.5: Distribution of the generic patches in hunks.

On the one hand, we recall that MINER regroups patches that are similar in terms of code context (AST) and repair action, but we do not consider the similarity among tokens. On the other hand, our current implementation of INFERRER[2] also considers the similarity of the tokens involved in the change to track a pattern: if the tokens involved in the set of patches can be abstracted in a single metavariable then a single transformation rule can be formed. Otherwise, the generic patch will include multiple transformation rules, each including specific tokens (e.g., specific method names). Figure 6.6 illustrates a concrete example of such a case. For both of the code examples, MINER produces the same AST rich edit script (cf. representation in Fig. 6.2) leading them to be placed in the same cluster. However, since they differ in terms of the tokens used (different method names and different parameters), INFERRER creates two distinct transformation rules.

```
−      scanf("%s", str);
+      gets(str);
```

```
−      error_setg_errno("%s: stat failed", fname);
+      error_setg_file_open(fname);
```

Figure 6.6: An example of patches sharing the same cluster but having distinct generic patches .

Note that distributions of generic patches in terms of the number of functions and functions also follow the same long tail shape: for example, we observed that ∼85% (=58 176 /68 368) of the patterns are inferred from a single function. We postulate that the distribution of locations can be used as a heuristic for prioritising pattern selection in the program (cf. RQ-4 for more insights).

[2] which is based on the algorithm of SPINFER [206]

> **RQ-2**: *Generic patches present a long tail distribution in terms of the number of code locations that were involved in their inference. FlexiRepair further provides tractability links to diagnose the code changes set that share similar transformations leading to a fix pattern.*

6.6.3 Repairability

We assess whether the inferred generic patches can be used to automate the generation of patches for real bugs. More specifically, we perform two program repair experiments by using the generic patches generated by Inferrer as the main input ingredients. Introclass and Codeflaws are leveraged for benchmarking.

Table 6.8 illustrates the comparative results in terms of numbers of plausible patches (i.e., that make the program pass all the test cases) for the black-box and white-box test suites. Among the selected 764 defects in Introclass, FlexiRepair can generate plausible patches for 186 defects using the black-box test suite and 261 plausible patches using the white-box test suite. Overall, we generate plausible patches for 288 defects of Introclass when both scenarios are combined. We compare the repair performance of FlexiRepair against 3 state-of-the-art APR tools which have been evaluated. With the white-box test suite, FlexiRepair ranks second in terms of the number of generated plausible patches, and third with the black-box scenario. It is noteworthy that FlexiRepair fixes significantly more bugs than other APR tools in some specific projects such as *checksum*, *grade*, and *syllables*.

Table 6.8: Number of Introclass bugs fixed by APR tools.

	FlexiRepair		GenProg		TrpAutoRepair		AE	
Project	WB	BB	WB	BB	WB	BB	WB	BB
checksum	23	23	3	8	1	0	1	0
digits	37	8	99	30	46	19	50	17
grade	12	8	3	2	2	2	2	2
median	44	27	63	108	36	93	16	58
smallest	75	56	118	120	118	119	92	71
syllables	70	64	6	19	9	14	5	11
Total	261	186	292	287	212	247	166	159

† The data about GenProg [117], TrpAutoRepair [189] and AE [236] are extracted from the experimental results reported by Le Goues *et al.* [116]

Table 6.9 lists example generic patches relevant for fixing Introclass defects. The first generic patch is an example of a fix pattern that matches several locations: it substitutes the *C* standard library function `scanf()` with `gets()`. According to documentation `scanf()` reads input until it encounters whitespace, newline or End Of File (EOF), whereas `gets()` reads input until it encounters newline or End Of File (EOF). We notice that *gets()* does not stop reading input when it encounters whitespace, but instead, it takes whitespace as a string, avoiding bugs. The other listed patterns are mostly related to control logic (i.e., wrong operator usage, boundary checks etc.) in `if` and `for` statements.

Our second experiment is performed on the Codeflaws benchmark. For this experiment, we limit the number of generic patches to Top-10 000 based on their frequency in code hunks. FlexiRepair can generate plausible patches for 83 defects using the test-suite I. 20 of those 83 defects have been validated to be correctly fixed using the test-suite II. We compare the repair performance of FlexiRepair against 5 state-of-the-art APR tools as illustrated in Table 6.10. There is an important performance gap between FlexiRepair and other state-of-the-art APR tools. We postulate that the limitation on the number of generic patches had a significant negative impact. Finding a good balance between efficiency (i.e., the search space must not explode by considering all possibilities)

Table 6.9: Selected generic patches fixing Introclass defects.

<table>
<tr><th>pattern</th><th>#defects</th><th>pattern</th><th>#defects</th></tr>
<tr><td>@@
expression E0;
@@
– scanf("%s", E0);
+ gets(E0);</td><td>83</td><td>@@
expression E0, E1;
@@
– if (E0 || E1)
– {
...
– }</td><td>14</td></tr>
<tr><td>@@
expression E1;
expression E0;
@@
– if (E0 && E1)
+ if (E1)
{
...
}
– else
+ else
{
...
}</td><td>45</td><td>@@
expression E3;
expression E2;
expression E1;
expression E0;
@@
– if (E0 < E1 && E2 > E3)
+ if (E0 <= E1 && E2 >= E3)
{
...
}</td><td>10</td></tr>
<tr><td>@@
expression E0, E1, E2;
@@
– for(E0 = E1;E0 < E2;E0++)
+ for(E0 = E1;E0 <= E2;E0++)
{
...
}</td><td>7</td><td>@@
expression E2;
expression E0;
binary operator B1 = {< ,>= };
@@
– if (E0 B1 E2)
+ if (E0 > E2)
{
...
}</td><td>6</td></tr>
</table>

Table 6.10: Number of Codeflaws bugs fixed by APR tools.

	FlexiRepair	Angelix	Prophet	SPR	GenProg	CoCoNuT
Total	20/83	318/591	301/839	283/783	[255-369]/1423	423/716

† In each column, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool. The data about Angelix [156], Prophet [140], SPR [138], GenProg [117] are extracted from the experimental results reported by CoCoNuT [144].

and effectiveness can be considered as an engineering detail. We discuss this in the following research question.

> **RQ-3**: *Note that we do not seek to outperform existing APR tools with our prototype implementation building on the generic patch specifications. Instead, we propose baseline performance for future research in template-based program repair that uses the proposed unified representation of fix patterns. Nevertheless, we note that the baseline is competitive with some state of the art on IntroClass benchmark.*

6.6.4 Efficiency

We assess the efficiency of repair in terms of Number of Patch Candidates (NPC) generated before the first plausible patch is found. NPC represent the invalid patches that an APR tool has consumed resources to test. NPC score has been advocated as a less biased metric of performance compared to execution time [35,60,134]. Our evaluation of the IntroClass benchmark distinguishes two categories:

1. **Nonsensical patches** are patches which cannot even make the patched buggy program successfully compile [88, 162].
2. **In-plausible patches** are patches which let the patched buggy program successfully compile, but fail to pass some test cases in the available test suite.

Figure 6.7 shows the distribution of the position of the first plausible patch when considering all sensical patches (i.e., patches that let the program compile). The median of the position for the black-box scenario is 23, and 31 for the white-box scenario. These represent NPC scores when considering only in-plausible patches.

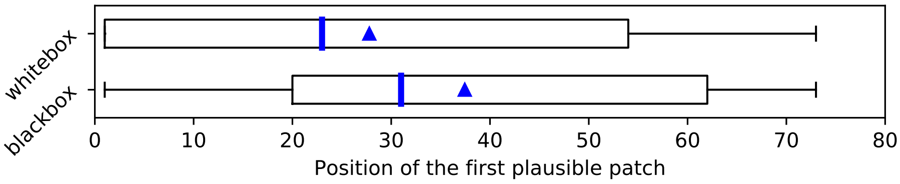


Figure 6.7: NPC of sensical patches.

Figure 6.8 shows the distribution of the position of the first plausible patch counting all the patches, i.e., including nonsensical patches). We note that the mean value of NPC in the black-box scenario is 41 626 and in the white-box scenario 31 587. Such high values indicate that the baseline tool applies first some generic patches that lead to nonsensical patches. Recall that, we select generic patches to apply first based on their frequency in code hunks in FlexiRepair. This result therefore suggests that other selection strategies could improve the overall results, and are thus worth to be explored extensively as future work.

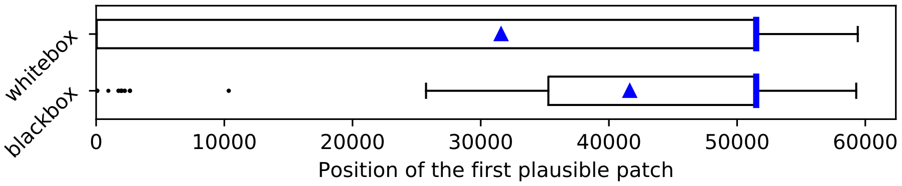


Figure 6.8: NPC of all patches.

We investigate a first simple strategy of generic patch selection for repair based on its recurrence in the mining dataset measured in terms of functions, files, patches, projects whose hunks contributed to the pattern abstraction. We experiment with all five cases and focus exclusively on the white-box scenario.

Figure 6.9 shows the corresponding distribution NPCs, excluding the nonsensical patches. The strategy where the selection is driven by the frequencies of the generic patches among the projects yields the best results: when we prioritise generic patches inferred from a large number of projects, the NPC score is lower.

Figure 6.10 illustrates the NPC considering all patches. Prioritising generic patches that have been inferred from a large number of projects or a large number of files leads to a significant reduction of NPC by ∼48% (from 31 587 to 16 501 when ordered by projects) and ∼57% (from 31 587 to 13 645 when ordered by files).

> **RQ-4**: *We note that the efficiency of* FlexiRepair *could be improved by better prioritizing generic patches. We show that because of the traceability in pattern inference, we are able to leverage frequency information to improve the efficiency score by halving the NPC.*

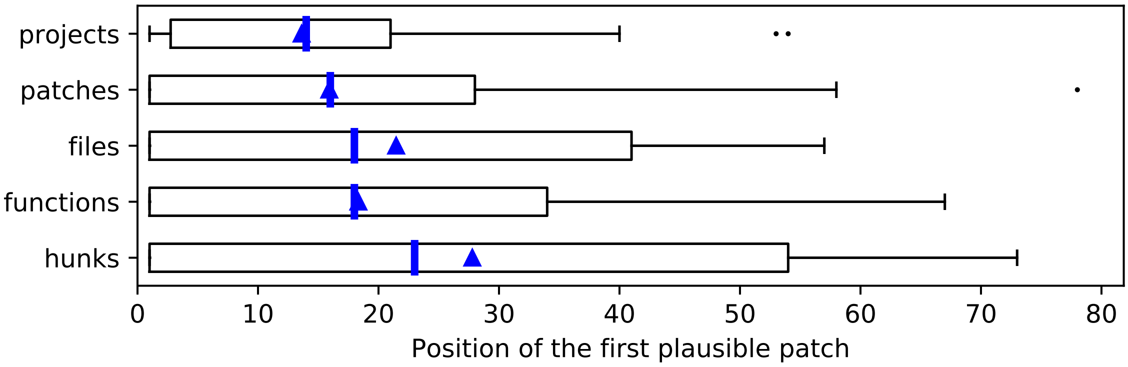


Figure 6.9: NPC of sensical patches for various selection strategies .

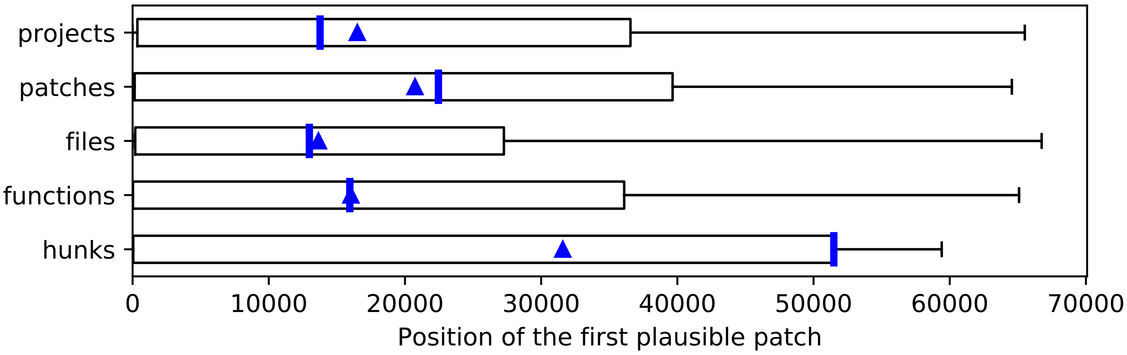


Figure 6.10: NPC of all patches for various selection strategies.

6.7 Summary

We have presented FlexiRepair, an open framework for template-based program repair where we build on the concept of generic patch to define a unified representation/notation for specifying fix patterns (aka templates). We show that generic patches are powerful for expressing fix patterns in a transparent and flexible way. FlexiRepair thus offers means, with a baseline, to measure and assess repair new contributions in template-based program repair (e.g., pattern inference, heuristics of candidate search, etc.). We evaluate the repair performance of a prototype implementation on the IntroClass and CodeFlaws benchmarks and we show that our baseline provides comparable performance to state of the art.

7 Conclusions and Future Work

In this chapter, we revisit the main contributions of this dissertation and present potential future research directions.

Contents

7.1 Conclusions & Future Work

In this dissertation, we presented ideas and techniques for boosting automated program repair towards its acceptability by developer communities. We have started with an empirical study on Linux kernel development project, and gather relevant insight on the practice of patching. Concretely, we have identified three distinct patching processes that are commonly used: (1) patches crafted entirely manually to fix bugs, (2) patches that are derived from warnings of bug detection tools, and (3) patches that are automatically generated based on fix patterns. The output of this study yielded several findings about i) the acceptance of patches ii) stability of the patches iii) on the nature of bugs being fixed iv) opportunities for improving automated repair techniques in production environments. We leveraged these findings to define the following research directions towards devising practical automated repair approaches.

7.1.1 Mining software repositories

In the first part, we focused on mining software repositories towards understanding code change properties and how they could be leveraged to guide program repair. To that end, we have presented `FixMiner`, a systematic and automated approach to mine relevant and actionable fix patterns for automated program repair. In `FixMiner`, we exploit the recurrences of the code changes to discover generic change patterns. The approach builds on a three-fold clustering strategy where we iteratively discover recurrent changes preserving the surrounding code context. We have evaluated `FixMiner` on thousands of software patches collected from open source projects. Preliminary results show that we are able to mine accurate patterns, efficiently exploiting change information. We also demonstrated the consistency of the mined patterns with the patterns in the literature. Finally, we integrated the mined patterns to an automated program repair prototype, $\texttt{PAR}_{\texttt{FixMiner}}$, with which we are able to correctly fix 26 bugs of the Defects4J benchmark. Beyond this quantitative performance, we show that the mined fix patterns are sufficiently relevant to produce patches with a high probability of correctness: 81% of $\texttt{PAR}_{\texttt{FixMiner}}$'s generated plausible patches are correct.

With `FixMiner`, we have demonstrated that mining software repositories could be helpful to guide program repair. In this domain, there are promising research directions that I plan to explore:

- i) Exploring code embedding approaches towards learning a deeper semantic representation of code in order to more accurately and efficiently reason about the recurrence of code changes. The aim will be to build an APR-adapted representation of code that will be leveraged to associate the change intention to various artefacts such as bug reports, test cases.
- ii) Guiding template selection based on categories of bug types that are sharing a common semantic representation. The objective that I seek is to ensure that similar bugs are patched with specific templates that are semantically relevant to bug types. This targeted selection strategy would facilitate early-stop in patch generation, lead to avoid a search space explosion while improving the patch correctness ratios, given the reasonable assumption that similar bugs are indeed fixed with similar patches.

7.1.2 Communication channels in software development

For the second part, we focused on communication channels in software development in order to assess to what extent they could be relevant in a real-world program repair scenario. We assumed that bug tracking systems constitute an essential artefact for guiding program repair as they contain the execution scenarios that were being carried out and the unexpected outcomes. Towards devising a program repair system driven by bug reports, first, we extensively study the performance of state-of-the-art bug localisation tools. Our empirical study of bug localisation tools suggests that each tool appears to be more successful than others in some specific regions of the datasets while another large region seems to be localisable similarly by all tools. We assumed that these regions contain features for sets of bug reports/code files pairs where the computed weights are effective. Building on this finding, we devise a file-level Information Retrieval based fault localisation (IRFL) tool, D&C. D&C builds a learning approach that is adaptively computing the most effective weights to apply to the similarity scores of IR features of a given pair of bug report/source code file. We explore such an approach with a supervised learning technique where classification is built by learning from the regions that appear to be successful with specific types of features. Comparison against the state-of-the-art shows that D&C provides a substantial performance improvement of MAP and MRR over all tools: MAP is improved by between 4 and up to 10 percentage points, while MRR is improved by between 1 and up to 12.

Building on D&C we propose to investigate a new repair pipeline, iFixR, that is driven by bug reports. iFixR adapts to the constraints of test cases unavailability when users report bugs. The proposed system revisits the fundamental steps, notably fault localisation, patch generation and patch validation, which are all tightly-dependent on the positive test cases in a test-based APR system : (1) bug reports are fed to an IR-based fault localiser; (2) patches are generated from fix patterns and validated via regression testing; (3) a prioritised list of generated patches is proposed to developers. Without making any assumptions on the availability of test cases, iFixR can generate and recommend priority correct (and more plausible) patches for a diverse set of user-reported bugs.

With iFixR and D&C, we have shown that bug reports could be handled automatically for a variety of bugs. This is an opportunity for issue trackers to add a recommendation layer to the bug triaging process by integrating patch generation techniques. There are, however, several directions I plan to explore for further investigation, among which:

- i) Automated classification of bug reports in order to associate the bug reports to a corresponding semantic (i.e., bug type). The objective of this classification is to allow the automation of repair with a template that is sharing a common semantic with a bug.
- ii) Tackling the vocabulary mismatch problem in IR-based fault localisation techniques as words in natural language (i.e., bug report) do not use the same lexicon as code tokens, thus preventing IR techniques from yielding good results. The goal is to propose new methods for learning and using embeddings of token pairs that implicitly learn relations between natural language words in bug reports and code tokens.
- iii) Learning to predict whether the given bug report is a suitable target for automated bug localisation towards improving the performance as well as the practical usage of bug localisation. We postulate that some bug reports may not be suitable for automation (i.e., they may be of low quality, incomplete, contain attachments (i.e., image) that cannot be interpreted by IR techniques) thus, our idea is to learn which bug reports could be localised and solely target these for automated bug localisation.

7.1.3 Generic concepts of patching

For the last part of the dissertation, we focused on exploring generic concepts of patching in the literature for establishing a common foundation for program repair pipelines. We assumed that building on top of well-accepted software maintenance concepts would facilitate the adoption of APR tools by practitioners. To that end, we proposed to build a patch generation system around the concept of generic patch whose underlying definition and structure is borrowed from the Linux community toolbox. We have presented FlexiRepair, an open framework for template-based program repair where we built on the concept of generic patch to define a unified representation/notation for specifying fix patterns (aka templates). With FlexiRepair we aim to separate implementation details from actual scientific contributions by providing an open, transparent and flexible repair pipeline on top of which all advancements in terms of efficiency, efficacy and usability can be measured and assessed rigorously.

Preliminary experiments with a prototype FlexiRepair on the IntroClass and CodeFlaws benchmarks suggest that it already constitutes a solid baseline with comparable performance to some of state of the art. For the future research in this direction, we believe it may be worthwhile to drain some research effort into building an automatic patch generation system that is based on robust, flexible and tractable techniques for boosting the acceptability by developer communities.

List of papers and tools

Papers included in this dissertation:

- A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017
- A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv preprint arXiv:1902.02703*, 2019
- A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020
- A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon. 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*, pages 314–325. ACM, 2019

Papers not included in this dissertation:

- K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon. A closer look at real-world patches. In *Proceedings of the 34th ICSME*, pages 275–286. IEEE, 2018
- K. Liu, K. Anil, K. Kim, D. Kim, and T. F. Bissyandé. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th APSEC*, pages 658–662. IEEE, 2018
- K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th ICST*, pages 102–113. IEEE, 2019
- K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th SANER*, pages 1–12. IEEE, 2019
- K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ICSE*, pages 1–12. IEEE, 2019
- K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. TBar : Revisiting template-based automated program repair. In *Proceedings of the 28th ISSTA*. ACM, 2019
- K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 615–627. ACM, 2020
- H. Tian, K. Liu, A. K. Kaboreé, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. Evaluating reprepentation learning of code changes for predicting patch correctness in program repair. *arXiv preprint arXiv:2008.02944*, 2020

Bibliography

[1] D&c. `https://github.com/d-and-c/d-and-c`, 2019.

[2] Ntlk framework. `https://www.nltk.org/`, 2019.

[3] R. Abreu, A. J. Van Gemund, and P. Zoeteweij. On the accuracy of spectrum-based fault localization. In *Proceedings of TAICPART-MUTATION*, pages 89–98. IEEE, 2007.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. *JSS*, 82(11):1780–1792, 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th ASE*, pages 88–99. IEEE, 2009.

[6] A. Afzal, M. Motwani, K. Stolee, Y. Brun, and C. Le Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering*, 2019.

[7] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[8] R. Al-Ekram, A. Adma, and O. Baysal. diffx: an algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[9] J. Andersen and J. L. Lawall. Generic patch inference. *Automated software engineering*, 17(2):119–148, 2010.

[10] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo. Semantic patch inference. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 382–385. IEEE, 2012.

[11] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th ICSE*, pages 361–370. ACM, 2006.

[12] J. Bader, A. Scott, M. Pradel, and S. Chandra. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

[13] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–317, New York, NY, USA, 2014. ACM.

[14] K. Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[15] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their ides. In *Proceedings of the 10th FSE*, pages 179–190. ACM, 2015.

[16] S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th ICSE*, pages 60–70. ACM, 2018.

[17] S. Bhatia and R. Singh. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129*, 2016.

[18] P. Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005.

[19] T. Bissyande. *Contributions for improving debugging of kernel-level services in a monolithic operating system*. PhD thesis, Université Sciences et Technologies-Bordeaux I, 2013.

[20] T. F. Bissyandé. Harvesting fix hints in the history of bugs. *arXiv preprint arXiv:1507.05742*, 2015.

[21] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 188–197. IEEE, 2013.

[22] T. F. Bissyandé, L. Réveillère, J. L. Lawall, and G. Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 60–69. IEEE, 2012.

[23] T. F. Bissyandé, L. Réveillère, J. L. Lawall, and G. Muller. Ahead of time static analysis for automatic generation of debugging interfaces to the linux kernel. *Automated Software Engineering*, pages 1–39, 2014.

[24] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere. Empirical evaluation of bug linking. In *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, pages 89–98, Washington, DC, USA, 2013. IEEE Computer Society.

[25] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[26] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th FSE*, pages 117–128. ACM, 2017.

[27] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522. ACM, 2017.

[28] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 114–126, New York, NY, USA, 2009. ACM.

[29] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 33–42, New York, NY, USA, 2010. ACM.

[30] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[31] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012.

[32] O. Chaparro, J. M. Florez, and A. Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 376–387. IEEE, 2017.

[33] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.

[34] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[35] L. Chen, Y. Pei, and C. A. Furia. Contract-based program repair without the contracts. In *Proceedings of the 32nd ASE*, pages 637–647. IEEE, 2017.

[36] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2019.

[37] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 243–247. IEEE, 2009.

[38] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[39] Z. Coker and M. Hafiz. Program transformations to fix c integers. In *Proceedings of the 35th ICSE*, pages 792–801. IEEE/ACM, 2013.

[40] V. Csuvik, D. Horváth, F. Horváth, and L. Vidács. Utilizing source code embeddings to identify correct patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 18–25. IEEE, 2020.

[41] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. IEEE Computer Society, 2009.

[42] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[44] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 47–56. IEEE, 2008.

[45] A. Duley, C. Spandikow, and M. Kim. Vdiff: a program differencing algorithm for verilog hardware description language. *Automated Software Engineering*, 19(4):459–490, 2012.

[46] S. T. Dumais. Latent semantic analysis. *Annual review of information science and technology*, 38(1):188–230, 2004.

[47] T. Durieux, R. Abreu, M. Monperrus, T. F. Bissyandé, and L. Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

[48] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th SANER*, pages 349–358. IEEE, 2017.

[49] T. Durieux and M. Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the IEEE/ACM 11th International Workshop in Automation of Software Test*, pages 85–91. IEEE, 2016.

[50] L. T. et al. Git. `http://git-scm.com/`, Last Accessed: Feb. 2017.

[51] J.-R. Falleri. GumTree. `https://github.com/GumTreeDiff/gumtree`, Last Access: Mar. 2018.

[52] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 313–324. ACM, 2014.

[53] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceeding of the 19th ICSM*, pages 23–32. IEEE, 2003.

[54] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 35–45. IEEE, 2006.

[55] B. Fluri, E. Giger, and H. C. Gall. Discovering patterns of change types. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 463–466, L'Aquila, Italy, 2008. IEEE.

[56] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11), 2007.

[57] P. Fournier-Viger, C.-W. Wu, A. Gomariz, and V. S. Tseng. VMSP: Efficient Vertical Mining of Maximal Sequential Patterns. In *Advances in Artificial Intelligence*, pages 83–94. Springer, Cham, May 2014.

[58] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*, volume 331. Prentice Hall Englewood Cliffs, NJ, 1992.

[59] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187, New York, NY, USA, 2012. ACM.

[60] A. Ghanbari, S. Benton, and L. Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 19–30. ACM, 2019.

[61] E. Greengrass. Information retrieval: A survey. 2000.

[62] S. Gulwani, I. Radiček, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices*, 53(4):465–480, 2018.

[63] R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing common c language errors by deep learning. In *Proceedings of the 31st AAAI*, pages 1345–1351. AAAI Press, 2017.

[64] Q. Hanam, F. S. d. M. Brito, and A. Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–156. ACM, 2016.

[65] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[66] M. Hashimoto and A. Mori. Diff/ts: A tool for fine-grained structural change analysis. In *2008 15th Working Conference on Reverse Engineering*, pages 279–288. IEEE, 2008.

[67] H. Heberle, G. V. Meirelles, F. R. da Silva, G. P. Telles, and R. Minghim. Interactivenn: a web-based tool for the analysis of sets through venn diagrams. *BMC bioinformatics*, 16(1):169, 2015.

[68] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13*, pages 121–130, San Francisco, CA, USA, 2013. IEEE.

[69] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the 22nd ASE*, pages 34–43. ACM, 2007.

[70] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106, 2004.

[71] S. Hu, Y. Liang, L. Ma, and Y. He. Msmote: improving classification performance when training data is imbalanced. In *Proceedings of the Second International Workshop on Computer Science and Engineering*, volume 2, pages 13–17. IEEE, 2009.

[72] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th ICSE*, pages 12–23. ACM, 2018.

[73] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao. Cldiff: generating concise linked code differences. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 679–690. ACM, 2018.

[74] A. Israeli and D. G. Feitelson. The Linux kernel as a case study in software evolution. *Journal of Systems and Software*, 83(3):485–501, 2010.

[75] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[76] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ISSTA*, pages 298–309. ACM, 2018.

[77] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th ASE*, pages 273–282. ACM, 2005.

[78] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[79] R. Just, C. Parnin, I. Drosos, and M. D. Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 287–297. ACM, 2018.

[80] N. J. Juzgado, A. M. Moreno, and W. Strigel. Guest editors' introduction: Software testing practices in industry. *IEEE Software*, 23(4):19–21, 2006.

[81] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. MintHint: Automated Synthesis of Repair Hints. In *Proceedings of the 36th International Conference on Software Engineering*, pages 266–276, New York, NY, USA, 2014. ACM.

[82] W. B. A. Karaa and N. Gribâa. Information retrieval with porter stemmer: a new version for english. In *Advances in computational science, engineering and information technology*, pages 243–254. Springer, 2013.

[83] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. LightGBM: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157, 2017.

[84] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (t). In *Proceedings of the 30th ASE*, pages 295–306. IEEE, 2015.

[85] S. W. Kent. Dynamic error remediation: A case study with null pointer exceptions. *University of Texas Master's Thesis*, 2008.

[86] L. Kernel. Bugzilla tracking system. `https://bugzilla.kernel.org`, Last Accessed: Feb. 2017.

[87] S. Khatiwada, M. Tushev, and A. Mahmoud. Just enough semantics: an information theoretic approach for ir-based software bug localization. *Information and Software Technology*, 93:45–57, 2018.

[88] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pages 802–811. IEEE, 2013.

[89] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[90] M. Kim and D. Notkin. Program Element Matching for Multi-version Program Analyses. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 58–64, New York, NY, USA, 2006. ACM.

[91] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319. IEEE Computer Society, 2009.

[92] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, volume 7, pages 333–343. Citeseer, 2007.

[93] S. Kim, K. Pan, and E. Whitehead Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45. ACM, 2006.

[94] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE'07)*, pages 489–498. IEEE, 2007.

[95] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th QRS*, pages 103–112. IEEE, 2013.

[96] A. N. Kolmogorov and S. V. Fomin. *Elements of the Theory of Functions and Functional Analysis*. Dover Publications, Mineola, NY, dover books on mathematics edition edition, Feb. 1999.

[97] A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Impact of tool support in patch construction. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 237–248. ACM, 2017.

[98] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon. D&c: A divide-and-conquer approach to ir-based bug localization. *arXiv preprint arXiv:1902.02703*, 2019.

[99] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, pages 1–45, 2020.

[100] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon. 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*, pages 314–325. ACM, 2019.

[101] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1 edition, Dec. 1992.

[102] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 61–72, New York, NY, USA, 2016. ACM.

[103] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *Proceedings of the 30th International Conference on Automated Software Engineering*, pages 476–481. IEEE, 2015.

[104] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Bug Localization with Combination of Deep Learning and Information Retrieval. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 218–229. IEEE, 2017.

[105] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *2018 USENIX Annual Technical Conference*, pages 601–614, 2018.

[106] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. Wysiwib: A declarative approach to finding api protocols and bugs in linux code. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 43–52, June 2009.

[107] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31th International Conference on Machine Learning*, pages 1188–1196. JMLR.org, 2014.

[108] T.-D. B. Le, F. Thung, and D. Lo. Predicting effectiveness of ir-based bug localization techniques. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*, pages 335–345. IEEE, 2014.

[109] T.-D. B. Le, F. Thung, and D. Lo. Will this localization tool be effective for this bug? mitigating the impact of unreliability of information retrieval based bug localization tools. *Empirical Software Engineering*, 22(4):2237–2279, 2017.

[110] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th FSE*, pages 593–604. ACM, 2017.

[111] X.-B. D. Le, Q. L. Le, D. Lo, and C. Le Goues. Enhancing automated program repair with deductive verification. In *Proceedings of the 32nd ICSME*, pages 428–432. IEEE, 2016.

[112] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues. Overfitting in semantics-based automated program repair. *EMSE Journal*, pages 1–27, 2018.

[113] X. D. Le, L. Bao, D. Lo, X. Xia, and S. Li. On reliability of patch correctness assessment. In *Proceedings of the 41st ICSE*, 2019.

[114] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *Proceedings of the 23rd SANER*, volume 1, pages 213–224. IEEE, 2016.

[115] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th ICSE*, pages 3–13. IEEE, 2012.

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

[117] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[118] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Commun. ACM*, 2019.

[119] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Proceedings of the 15th TACAS*, pages 292–306. Springer, 2009.

[120] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon. Bench4bl: reproducibility study on the performance of IR-based bug localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 61–72. ACM, 2018.

[121] Y. Li, S. Wang, and T. N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 602–614, 2020.

[122] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 26th PLDI*, pages 15–26. ACM, 2005.

[123] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[124] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu. An empirical study on the characteristics of python fine-grained source code change types. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 188–199. IEEE, 2016.

[125] LIP6. Coccinelle. `http://coccinelle.lip6.fr/`, Last Accessed: Feb. 2017.

[126] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of the 6th ICST*, pages 282–291. IEEE, 2013.

[127] K. Liu, K. Anil, K. Kim, D. Kim, and T. F. Bissyandé. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 25th APSEC*, pages 658–662. IEEE, 2018.

[128] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st ICSE*, pages 1–12. IEEE, 2019.

[129] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon. Mining fix patterns for findbugs violations. *TSE*, 2018.

[130] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon. A closer look at real-world patches. In *Proceedings of the 34th ICSME*, pages 275–286. IEEE, 2018.

[131] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th ICST*, pages 102–113. IEEE, 2019.

[132] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th SANER*, pages 1–12. IEEE, 2019.

[133] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. TBar : Revisiting template-based automated program repair. In *Proceedings of the 28th ISSTA*. ACM, 2019.

[134] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 615–627. ACM, 2020.

[135] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *Proceedings of the 25th SANER*, pages 118–129. IEEE, 2018.

[136] B. Livshits and T. Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[137] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th FSE*, pages 727–739. ACM, 2017.

[138] F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 10th FSE*, pages 166–178. ACM, 2015.

[139] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 702–713. IEEE, 2016.

[140] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd POPL*, pages 298–312. ACM, 2016.

[141] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 49, pages 227–238. ACM, 2014.

[142] L. LUCIA, F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *Proceedings of the 9th MSR*, pages 74–77, 2012.

[143] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[144] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 101–114, 2020.

[145] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. Bears: An extensible java bug benchmark for automatic program repair studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478. IEEE, 2019.

[146] C. D. Manning, C. D. Manning, and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[147] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. Slice-based statistical fault localization. *JSS*, 89:51–62, 2014.

[148] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, 2019.

[149] M. Martinez, L. Duchien, and M. Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 388–391. IEEE, 2013.

[150] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[151] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[152] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th ISSTA*, pages 441–444. ACM, 2016.

[153] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *Proceedings of the 10th SSBSE*, pages 65–86. Springer, 2018.

[154] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th ICSE*, pages 298–309. ACM, 2018.

[155] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*, pages 448–458. IEEE, 2015.

[156] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701. ACM, 2016.

[157] N. Meng, M. Kim, and K. S. McKinley. Sydit: creating and applying a program transformation from an example. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 440–443, 2011.

[158] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices*, 46(6):329–342, 2011.

[159] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 502–511. ACM, 2013.

[160] G. A. Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[161] T. Molderez, R. Stevens, and C. De Roover. Mining change histories for unknown systematic edits. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 248–256. IEEE Press, 2017.

[162] M. Monperrus. 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 *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242, 2014.

[163] M. Monperrus. Automatic software repair: a bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24, 2018.

[164] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[165] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus. On the relationship between the vocabulary of bug reports and source code. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pages 452–455. IEEE, 2013.

[166] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE, 2014.

[167] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[168] M. Nayrolles and A. Hamou-Lhadj. Clever: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 153–164, 2018.

[169] I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[170] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272. ACM, 2011.

[171] H. A. Nguyen, A. T. Nguyen, and T. N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 138–147. IEEE, 2013.

[172] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pages 772–781. IEEE, 2013.

[173] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 315–324, 2010.

[174] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. In *Proceedings of the IEEE 18th International Conference on Program Comprehension*, pages 68–71. IEEE, 2010.

[175] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 343–347. IEEE, 2014.

[176] H. Oumarou, N. Anquetil, A. Etien, S. Ducasse, and K. D. Taiwe. Identifying the exact fixing actions of static rule violation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 371–379. IEEE, 2015.

[177] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of 3rd EuroSys*, volume 42, pages 247–260. ACM, 2008.

[178] N. Palix, J.-R. Falleri, and J. Lawall. Improving pattern tracking with a language-aware tree differencing algorithm. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 43–52, Mar. 2015.

[179] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: Ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 305–318, Newport Beach, California, USA, 2011.

[180] N. Palix, G. Thomas, S. Saha, C. Calvès, G. Muller, and J. Lawall. Faults in Linux 2.6. *ACM Trans. Comput. Syst.*, 32(2):4:1–4:40, June 2014.

[181] K. Pan, S. Kim, and E. J. Whitehead. Toward an understanding of bug fix patterns. *EMSE Journal*, 14(3):286–315, 2009.

[182] M. Papadakis and Y. Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[183] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 40–49. IEEE Press, 2012.

[184] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th ISSTA*, pages 199–209. ACM, 2011.

[185] M. Pawlik and N. Augsten. Rted: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[186] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the 39th ICSE*, pages 609–620. IEEE/ACM, 2017.

[187] J. Petrić, T. Hall, and D. Bowes. How effectively is defective code actually tested?: An analysis of junit tests in seven open source systems. In *Proceedings of the 14th PROMISE*, pages 42–51. ACM, 2018.

[188] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[189] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.

[190] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. Technical report, Massachussets Institute of Technology, 2015.

[191] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 322–331, 2011.

[192] M. M. Rahman and C. K. Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 621–632. ACM, 2018.

[193] S. Rao and A. Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 43–52. ACM, 2011.

[194] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 30–39. IEEE, 2003.

[195] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th ICSE*, pages 404–415. IEEE/ACM, 2017.

[196] R. Rolim, G. Soares, R. Gheyi, and L. D'Antoni. Learning quick fixes from code repositories. *arXiv preprint arXiv:1803.03806*, 2018.

[197] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th MSR*, pages 10–13. IEEE, 2018.

[198] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. On the effectiveness of information retrieval based bug localization for c programs. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 161–170. IEEE, 2014.

[199] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 345–355. IEEE, 2013.

[200] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 32nd ASE*, pages 648–659. IEEE, 2017.

[201] G. Salton and M. J. McGill. Introduction to Modern Information Retrieval. 1986.

[202] G. Salton, A. Wong, and C.-S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[203] A. Schröter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, pages 118–121. IEEE, 2010.

[204] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, 2014.

[205] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to Information Retrieval*, volume 39. Cambridge University Press, 2008.

[206] L. Serrano, V.-A. Nguyen, F. Thung, L. Jiang, D. Lo, J. Lawall, and G. Muller. SPINFER: Inferring semantic patches for the linux kernel. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 235–248. USENIX Association, July 2020.

[207] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM.

[208] B. Sisman and A. C. Kak. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 309–318. IEEE Press, 2013.

[209] S. S. Skiena. The stony brook algorithm repository. *URL http://www. cs. sunysb. edu/algorith/implement/nauty/implement. shtml*, 1997.

[210] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[211] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*, 2018.

[212] F. Song and W. B. Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, pages 316–321. ACM, 1999.

[213] M. Soto and C. Le Goues. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th SANER*, pages 221–231. IEEE, 2018.

[214] J. Spaans. The linux kernel mailing list, Last Accessed: Feb. 2017. `http://lkml.org`.

[215] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[216] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 22nd ISSTA*, pages 314–324. ACM, 2013.

[217] Synopsys. Coverity. `http://www.coverity.com/`, Last Accessed: Feb. 2017.

[218] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki. A Preliminary Study on Using Code Smells to Improve Bug Localization. In *Proceedings of the 26th International Conference on Program Comprehension*, pages 324–327. ACM, 2018.

[219] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 471–482. IEEE Press, 2015.

[220] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 180–182. IEEE Press, 2017.

[221] Y. Tao, J. Kim, S. Kim, and C. Xu. Automatically generated patches as debugging aids: a human study. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 64–74. ACM, 2014.

[222] Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190. IEEE, 2015.

[223] S. W. Thomas, M. Nagappan, D. Blostein, and A. E. Hassan. The impact of classifier configuration and classifier combination on bug localization. *IEEE Transactions on Software Engineering*, 39(10):1427–1443, 2013.

[224] H. Tian, K. Liu, A. K. Kaboreé, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. Evaluating reprepentation learning of code changes for predicting patch correctness in program repair. *arXiv preprint arXiv:2008.02944*, 2020.

[225] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*, pages 386–396. IEEE Press, 2012.

[226] R. v. Tonder and C. L. Goues. Defending against the attack of the micro-clones. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4, May 2016.

[227] L. Torvalds. Linux kernel git tree. `http://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/`, Last Accessed: Feb. 2017.

[228] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837. ACM, 2018.

[229] Y. Ueda, T. Ishio, A. Ihara, and K. Matsumoto. Devreplay: Automatic repair with editable fix pattern. *arXiv preprint arXiv:2005.11040*, 2020.

[230] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus. How to design a program repair bot?: insights from the repairnator project. In *Proceedings of the 40th ICSE*, pages 95–104. ACM, 2018.

[231] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2015.

[232] S. Wang and D. Lo. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63. ACM, 2014.

[233] S. Wang and D. Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[234] S. Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 171–180. IEEE, 2014.

[235] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th ISSTA*, pages 61–72. ACM, 2010.

[236] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366, Silicon Valley, CA, USA, 2013. IEEE.

[237] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE, 2009.

[238] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st ICSE*, pages 364–374. IEEE, 2009.

[239] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 231–240. IEEE, 2006.

[240] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172*, 2017.

[241] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th ICSE*, pages 1–11. IEEE/ACM, 2018.

[242] M. Wen, R. Wu, and S.-C. Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 262–273. IEEE, 2016.

[243] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 26th SANER*. IEEE, 2019.

[244] S. Wiki. Sparse. `https://sparse.wiki.kernel.org`, Last Accessed: Feb. 2017.

[245] Wikipedia. Benevolent dictator for life. `http://en.wikipedia.org/wiki/Benevolent_dictator_for_life`, Last Accessed: Feb. 2017.

[246] W. E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. 1990.

[247] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 181–190. IEEE, 2014.

[248] S. M. Wong, W. Ziarko, and P. C. Wong. Generalized vector spaces model in information retrieval. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 18–25. ACM, 1985.

[249] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *TSE*, 42(8):707–740, 2016.

[250] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214. ACM, 2014.

[251] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *TOSEM*, 22(4):31:1–31:40, 2013.

[252] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ISSTA*, pages 226–236. ACM, 2017.

[253] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd ASE*, pages 660–670. IEEE/ACM, 2017.

[254] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th ICSE*, pages 789–799. ACM, 2018.

[255] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE, 2017.

[256] J. Xu and W. B. Croft. Cluster-based language models for distributed retrieval. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 254–261. ACM, 1999.

[257] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE*, 43(1):34–55, 2017.

[258] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the 30th ICSME*, pages 191–200. IEEE, 2014.

[259] B. Yang and J. Yang. Exploring the differences between plausible and correct patches at fine-grained level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–8. IEEE, 2020.

[260] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *Proceedings of the 11th FSE*, pages 831–841. ACM, 2017.

[261] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus. Automated classification of overfitting patches with statically extracted code features. *arXiv preprint arXiv:1910.12057*, 2019.

[262] X. Ye, R. Bunescu, and C. Liu. 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*, pages 689–699. ACM, 2014.

[263] X. Ye, R. Bunescu, and C. Liu. Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation. *IEEE Transactions on Software Engineering*, 42(4):379–402, 2016.

[264] S.-J. Yen and Y.-S. Lee. Cluster-based under-sampling approaches for imbalanced data distributions. *Expert Systems with Applications*, 36(3):5718–5727, 2009.

[265] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[266] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *STVR*, 22(2):67–120, 2012.

[267] K. C. Youm, J. Ahn, J. Kim, and E. Lee. Bug Localization Based on Code Change Histories and Bug Reports. In *Proceedings of the 2015 Asia-Pacific Software Engineering Conference*, pages 190–197. IEEE, 2015.

[268] K. C. Youm, J. Ahn, and E. Lee. Improved bug localization based on code change histories and bug reports. *IST*, 82:177–192, 2017.

[269] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. *arXiv preprint arXiv:1703.00198*, 2017.

[270] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empirical Software Engineering*, 24(1):33–67, 2019.

[271] Y. Yuan and W. Banzhaf. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 2018.

[272] R. Yue, N. Meng, and Q. Wang. A characterization study of repeated bug fixes. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 422–432. IEEE, 2017.

[273] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM, 2006.

[274] Z. Zhang, W. K. Chan, T. Tse, Y.-T. Yu, and P. Hu. Non-parametric statistical fault localization. *JSS*, 84(6):885–905, 2011.

[275] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 913–923. IEEE, 2015.

[276] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 14–24. IEEE, 2012.

[277] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss. What makes a good bug report? *TSE*, 36(5):618–643, 2010.