Leveraging Light-Weight Analyses to Aid Software Maintenance

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Zachary P. Fry

May 2014

Abstract

While software systems have become a fundamental part of modern life, they require maintenance to continually function properly and to adapt to potential environment changes [1]. Software maintenance, a dominant cost in the software lifecycle [2], includes both adding new functionality and fixing existing problems, or "bugs," in a system. Software bugs cost the world's economy billions of dollars annually in terms of system down-time and the effort required to fix them [3].

This dissertation focuses specifically on corrective software maintenance — that is, the process of finding and fixing bugs. Traditionally, managing bugs has been a largely manual process [4]. This historically involved developers treating each defect as a unique maintenance concern, which results in a slow process and thus a high aggregate cost for finding and fixing bugs. Previous work has shown that bugs are often reported more rapidly than companies can address them, in practice [5].

Recently, automated techniques have helped to ease the human burden associated with maintenance activities. However, such techniques often suffer from a few key drawbacks. This thesis argues that automated maintenance tools often target narrowly scoped problems rather than more general ones. Such tools favor maximizing local, narrow success over wider applicability and potentially greater cost benefit. Additionally, this dissertation provides evidence that maintenance tools are traditionally evaluated in terms of functional correctness, while more practical concerns like ease-of-use and perceived relevance of results are often overlooked. When calculating cost savings, some techniques fail to account for the introduction of new workflow tasks while claiming to reduce the overall human burden. The work in this dissertation aims to avoid these weaknesses by providing *fully automated*, *widely-applicable* techniques that both *reduce the cost* of software maintenance and meet relevant *human-centric quality* and *usability* standards.

This dissertation presents software maintenance techniques that reduce the cost of both finding and fixing bugs, with an emphasis on comprehensive, human-centric evaluation. The work in this thesis uses lightweight analyses to leverage latent information inherent in existing software artifacts. As a result, the associated techniques are both scalable and widely applicable to existing systems. The first of these techniques *clusters* closely-related, automatically generated defect reports to aid in the process of bug triage and repair. This clustering approach is complimented by an automatic *program repair* technique that generates and validates candidate defect patches by making sweeping optimizations to a

state-of-the-art automatic bug fixing framework. To fully evaluate these techniques, experiments are performed that show net cost savings for both the clustering and program repair approaches while also suggesting that actual human developers both agree with the resulting defect report clusters and also are able to understand and use automatically generated patches.

The techniques described in this dissertation are designed to address the three historically-lacking properties noted above: generality, usability, and human-centric efficacy. Notably, both presented approaches apply to many types of defects and systems, suggesting they are *generally applicable* as part of the maintenance process. With the goal of *comprehensive evaluation* in mind, this thesis provides evidence that humans both agree with the results of the techniques and could feasibly use them in practice. These and other results show that the techniques are *usable*, in terms of both minimizing additional human effort via full automation and also providing understandable maintenance solutions that promote continued system quality. By evaluating the associated techniques on programs spanning different languages and domains that contain thousands of bug reports and millions of lines of code, the results presented in this dissertation show potential concrete cost savings with respect to finding and fixing bugs. This work suggests the feasibility of further automation in software maintenance and thus increased reduction of the associated human burdens.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Zachary P. Fry

This dissertation has been read and approved by the Examining Committee:

Westley R. Weimer, Advisor

Worthy N. Martin, Committee Chair

Jack W. Davidson

Stephanie Forrest

Greg Gerling

Accepted for the School of Engineering and Applied Science:

James H. Aylor, Dean, School of Engineering and Applied Science

May 2014

We are stuck with technology when what we really want is just stuff that works.
— Douglas Adams

Acknowledgments

"Could it be semantics generating the mess we're in?"

– Michael Burkett

I must first thank my advisor, Westley Weimer, for the heaps of wisdom he has imparted to me throughout my tenure as his student. Wes has far exceeded my expectations as an advisor, consistently going above and beyond the usual duties to make me a well-rounded, effective researcher. Though his advising style is often not for the weak-hearted, I am endlessly thankful for the wisdom and patience Wes has provided me throughout the past six years.

I have been fortunate enough to have many sources of inspiration and research advice over the years. I'd like to thank Stephanie Forrest for her considerable help throughout my graduate research career. She has consistently provided an excellent perspective and ample help with many collaborative research projects and for that I am very thankful. Additionally, I'd like to express an immense amount of gratitude to both of my undergraduate advisors, Lori Pollock and K. Vijay-Shanker — without your help, I would not be where I am today. Lori and Vijay opened me up to research early on and helped get me started off on the right foot; for that I am very thankful. Additionally, I would like to thank David Shepherd and Emily Hill for acting as excellent research mentors throughout my undergraduate research career.

One could not have asked for a smarter and more enjoyable research group than the "WRG." Kinga, Pieter, Ray, Claire, Adam, and Jon have provided both insightful contributions to my work and also ample distraction when necessary. I thank them all for the entertaining and informative adventure.

Thank you finally to my family and friends — it has been a wonderful, but sometimes trying, ride through graduate school. You have all been a constant source of support and much appreciated diversion, when appropriate. Special thanks to Jeremy Lacomis for ample help proofreading this document.

Contents

List of Tables

List of Figures

List of Terms

Note: terms are linked to their associated glossary entry at their first introduction or definition, and again at their first use in each subsequent chapter. Within this glossary, terms are linked more comprehensively.

bug — "a defect that causes a reproducible or catastrophic malfunction." [6] See defect. ix–xii, 1, 14

bug report — a collection of pre-defined categorical fields, free-form text, and attachments, that describe the symptoms and, potentially, the causes of a bug. The goal of a bug report is to aid in the triage, management, and repair of a bug. Bug reports can be produced manually by humans or automatically using, for instance, static analysis. ix, xi, 10, 14, 20

code clone — code that was copied and then pasted (often verbatim) in different locations across a system. Code clones are commonly believed to be detrimental because if they contain defects, the debugging effort has to be duplicated for each instance. ix, 16, 29, 39

code quality metric — a heuristic, quantitative approach to measuring software quality, including both functional properties (e.g., lack of defects) and non-functional ones (e.g., readability or understandability). ix, 16, 74

comprehensive evaluation — in the context of software tools in this dissertation, in addition to more traditional quantitative or empirical assessments, a comprehensive evaluation might also consider more human-centric notions of value such as understandability and human perceived quality of results. ix, 3, 5, 8, 27, 47, 72, 93

debugging — the process of understanding a program's specification and implementation to find and repair a bug corresponding to a deviation between the two. ix, 23

defect — quoting directly: "A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldnt), causing a failure." [7, p. 11]. ix–xi, 1, 14

defect report — see bug report. ix, xii, 4, 21, 22, 28

error — in the context of software functionality, either a deviation between a system's implementation and its specification or its specification and requirements that leads to a behavioral failure that can manifest as a bug or defect. ix, x, xii, 1

expressive power — in the context of program repair, a measure of the range of bug types a technique may theoretically be able to fix. Expressive power relates to the types and scopes of the bugs an approach can consider as well as the natures and granularities of program changes it can make. A tool with more expressive power has the potential to fix a wider range of bug types. ix, 50, 65, 81

fault — in the context of software behavior, "a defect that causes a reproducible or catastrophic malfunction" [6]. ix, xii, 1, 14

fault localization — the task of identifying the specific code statement(s) responsible for a given bug or error. Fault localization is typically a precursor to repair. ix, 23, 57

fitness distance correlation — measures whether a fitness signal accurately models some ground-truth notion of how close a mutant is to the desired goal. It does so by correlating measured fitness with some notion of actual mutant quality. ix, 51

fitness function — the objective function in a genetic programming search algorithm, measuring the desirability of a candidate solution. Desirability in this context refers to how close a given candidate patch is to an eventual repair. ix, 51

formal method — a subset of static analysis techniques that use rigorous mathematical methods to prove certain properties about program behavior. ix, xi, 20

general — in the context of this dissertation, with respect to software tools, widely applying to a range of programs or bugs, ideally to increase the impact of such a technique in practice. ix, 2, 5, 7, 27, 40, 47, 65, 72

genetic algorithm — a stochastic search and optimization strategy that mimics the process of biological evolution. ix, x, 24, 51

genetic programming — the application of genetic algorithms to programs. In the context of this dissertation, the goal of genetic programming is to find programs that fix known defects. ix–xi

heuristic — an estimation technique that may not give an optimal answer to the given query, but can generally do so more quickly than more absolute approaches. ix, 49, 51, 70

mutant — in the context of genetic programming, corresponds to a set of chromosomal (i.e., programmatic) changes to some original program. Synonymous with a candidate patch in this dissertation. ix–xii, 18, 49

mutation — in the context of a genetic programming, an operator that modifies source code or a binary to generate a new candidate patch. Mutations can be performed on either the original program or an existing mutant.. ix, xi, 18, 49

mutation testing — a technique used to measure test suite adequacy whereby mutations are systematically introduced into a program to simulate real world bugs. An accompanying test suite is then run to measure how many of the seeded bugs are exposed. Ideally, a test suite that recognizes many seeded bugs might also adequately expose unknown bugs. ix, 8, 18, 49, 67

natural language — in the context of this dissertation, English language inherent in software artifacts (e.g., source code or bug reports). This information often embodies human choice and can thus be analyzed to infer human intuition. ix, 6, 29, 34

patch — a set of changes to a program source code or binary, aimed at fixing a specific bug. We make a distinction between a potential, unverified candidate patch and one that actually fixes the bug (i.e., a validated repair). ix–xi, 4, 14, 23, 49, 77

program equivalence — is a judgment that determines if the runtime behavior of two or more programs is identical. The problem of deciding program equivalence is provably undecidable, but can be conservatively approximated using formal semantics or data flow analyses. ix, 19, 49, 58

program repair — the act of creating a valid patch for a given defect, either automatically or manually. ix, 4, 23, 48

repair — a validated candidate patch that fixes a given bug. Validation can consist of verification via formal methods or manual inspection as well as rigorous testing. ix–xi, 49

search space — the collection of all possible considerable options for a given search algorithm. In the context of program repair the search space corresponds to all programs that can be created given a set of possible mutations. Ideally, the search space contains a valid repair. ix, 18, 50, 57, 60

similarity metric — quantitatively measures similarity between two entities. In the context of this dissertation we measure similarity between natural language artifacts using metrics that take into account the order or frequency of tokens in a string. The axioms of metric spaces (i.e., non-negativity, coincidence, symmetry, and the triangle inequality) do not always hold for language metrics of this type. ix, 33

software evolution — encompasses all activities throughout both software development and software maintenance. ix, xii, 1, 25

software maintenance — the portion of the software evolution process comprising all activity after project deployment. This includes both adding new features and fixing bugs. ix, xii, 1, 13

static analysis — a class of methods for reasoning about the runtime behavior of a program without actually executing it. Static analyses often involve building knowledge about a program through, for instance, analyzing its formal semantics and control flow. ix, x, 3, 19, 28

test case — following Binder *et al.* [8], the combination of a program input, the expected programmatic output for that input, and an oracle comparator that can validate whether the observed output matches the expected output. Test cases can be generated manually (by humans) or automatically (via systematic, computational methods). ix, xii, 5, 17, 24, 49

test suite — a collection of test cases. ix, xi, 17, 24, 49, 77

test suite prioritization — a technique for reducing the cost of testing by exposing faults in the least amount of time. This may be accomplished, for instance, by either running high-impact tests first or by favoring tests that are likely to fail early. ix, 18, 49

testing — execution of one or more test cases to attempt to either expose or rule out faults. The goal of testing is to gain confidence that a program's implementation matches its specification. ix, xi, 1, 17, 49

triage — in the context of software maintenance, and specifically bug reporting, the process of reviewing a defect report to determine whether it contains sufficient and suitable information to recreate and verify the underlying bug. The process often also includes assigning the bug a priority and a developer to further investigate and fix the underlying error. ix, 21, 22, 29

usable — in the context of this dissertation, being easy for humans to employ and operate while also requiring minimal additional human effort. ix, 3, 5, 10, 27, 40, 47, 72

variant — in the context of this dissertation, synonym of mutant. ix, 49

Chapter 1

Introduction

> "Even the best planning is not so omniscient as to get it right the first time."
>
> – *Fred Brooks [9]*

SOFTWARE is staggeringly pervasive in the modern world. Like its mechanical predecessors, software has proven to be an effective tool for solving important problems. However, it too requires maintenance to continue to function properly. Software maintenance and, more broadly, software evolution have long been recognized as crucial to the continued efficacy of programs in practice [1, 10]. Software maintenance is also a *dominant* cost associated with the lifecycle of modern systems [11]; it can account for up to 90% of the total cost of producing software [2]. Among other activities, the maintenance process includes adding new functionality and fixing existing problems in the system — colloquially, adding features and fixing bugs. In this dissertation, we will focus on the latter concern: corrective maintenance, which loosely consists of identifying, locating, and fixing software defects to ensure continued system quality. In this dissertation, we will henceforth use the terms "fault," "bug," and "defect" interchangeably to mean a coding error that leads to some manner of system failure deemed to be unacceptable for regular program execution.

Software systems ship with known and unknown bugs as a matter of practicality [12]. Given limited resources (a pervasive problem in software development processes), comprehensively testing a system can be prohibitively expensive [13]. As a result, corrective software maintenance is an important part of the software lifecycle to fix bugs that arise after deployment. Resource constraints not only lead to the introduction of bugs, but also prevent developers from addressing them all in a timely fashion [5]. In practice, the average lifespan of a bug can be on the order of weeks [14], months [15], or, surprisingly, years even for some high-priority bugs [16].

The evidence supporting our collective inability to keep up with the corrective software maintenance process is even more alarming when examining the impact and cost of such bugs. Even conceptually simple bugs can have far-

reaching impacts — the "Y2K" bug (i.e., two-digit date representations which caused ambiguity in the new millennium) reportedly required corrective maintenance in as many as 75% of software systems worldwide [17]. As an additional example, a single-line bug in the Microsoft Zune media player code caused the associated devices to freeze completely for 24 hours in 2008 [18]. Software bugs of this magnitude have broad impacts, spanning many important domains from commerce to medicine, governmental infrastructure to simple quality of life. With respect to concrete monetary cost, a 2002 survey estimates that every year software bugs in the US alone cost US$59.5 billion (0.6% of GDP) [19]. In 2013, researchers have estimated that bugs are costing the global economy as much as £192 billion annually [3], suggesting the monetary burden has not waned. A security-specific Norton study estimates the global cost of cybercrime as US$114 billion annually, with a further US$274 billion in lost time [20]. Software defects represent considerable operational and monetary burdens on both developers and end users of software systems.

The cost of addressing software bugs is due in part to the manual nature of many parts of the maintenance process. Critical software maintenance tasks currently require extensive human intervention or creativity, making them time-intensive and thus expensive. Lehman's Law of Conservation of Familiarity supports this principle; it states that all humans associated with a software system "must maintain mastery of its content and behavior to achieve satisfactory evolution." [21] The trend in software maintenance research has been towards bridging the gap between traditional manual strategies and partially automated processes. While considerable success has been achieved in this regard (see Chapter 2), certain maintenance processes remain largely manual — for instance, bug fixing is still largely carried out by hand in practice [4, 22].

1.1 Why state of the art software maintenance tools are inadequate

To mitigate the costs associated with software maintenance, extensive research has developed automated tools [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35] that reduce the human burden associated with the software maintenance process. We have observed, however, that such tools tend to suffer from three common problems in practice: a lack of generality, an absence of comprehensive evaluation, and low usability.

Generality. First, tools are often narrowly focused — in the domain of software maintenance, tools typically address only a single part of the process (e.g., only fixing certain types of bugs). Further, tools often only apply to a particular setting or constrained situation: this increases effectiveness but decreases wide applicability [26, 27, 36]. While such techniques are effective at their respective narrowly focused tasks, one would have to use a large, diverse set of such tools simultaneously to guard against a broad range of possible program failures. This necessarily adds steps to development workflows and requires additional knowledge and tool mastery to carry out maintenance tasks [21] which could increase cost. As such, we desire tools that are general in focus to foster wide applicability and possibly greater impact.

Comprehensive Evaluation. A second problem concerns human usability as it relates to software tools. Research in software maintenance often includes empirical evaluations with respect to other state-of-the-art research tools [37, 38, 39], but practical concerns like human usability and understandability are often overlooked. Anecdotally, this phenomenon may occur because measuring the human-based consequences of such tools can be both difficult and expensive [40] (e.g., setting up an experiment involving human subject feedback can be cumbersome), but we believe that a full, comprehensive evaluation of the efficacy of software maintenance tools must examine such pragmatic concerns [35, 41, 42, 43, 44].

Usability. Third and finally, with respect to usability, maintenance tools often promise net cost savings while requiring additional human input. This tradeoff can hamper adoption (i.e., by requiring users to add additional steps to their workflows, when the goal is often to remove such burdens) and obfuscates whether the tool does, in fact, yield a *net* cost savings. Consider one such tool, developed by Kremenek *et al.*, that attempts to identify related bugs based on their locations in the code base [38]. The tool requires iterative feedback from users, which is not accounted for in the cost savings model and represents additional effort for users in practice. Coverity, a commercial software tools company, echoes this concern by noting that when developing their static analysis bug finder, "as much as possible, we avoided using annotations or specifications to reduce manual labor." [22, p. 66] This viewpoint is widespread: a 2012 Microsoft study found that one of developers' main concerns with respect to software tools is "ease of use" [45], further suggesting that burdening users with additional input requirements may hamper a tool's widespread adoption. That study notes that tools often offer information counter to that deemed most useful by developers, suggesting that simply providing *any* additional information does not necessarily equate to usability [45]. We believe the utility of a tool should be measured directly, rather than assumed.

We will specifically address each of these three concerns throughout this dissertation and present practical solutions to concretely reduce the cost of software maintenance.

1.2 A approach to improving aspects of software maintenance

The work in this dissertation focuses specifically on reducing costs associated with software maintenance by providing automated improvements to two crucial tasks, bug finding and bug fixing, with the following high-level solution requirements:

1. **Generality.** We desire approaches that apply broadly to bugs of different types, severities, scopes, and domains. General techniques may be more widely applicable and thus represent larger cost savings by handling more bugs overall.

2. **Comprehensive evaluation.** In addition to traditional cost or performance-based evaluations, we desire evidence that the developed techniques are effective in terms of human-based metrics that apply to the software maintenance process in practice.

3. **Usability.** We desire techniques that require minimal human input, that can be effective "off-the-shelf," where applicable, to encourage widespread adoption and yield practical cost savings.

This dissertation outlines an end-to-end approach to reducing the cost of corrective software maintenance. In this introductory section, when appropriate, we will use analogies to building maintenance to ease the explanation of the underlying software maintenance tasks. Specifically, there are three main research thrusts we present in this document with respect to finding and fixing bugs:

- **Clustering similar automatically generated defect reports** — The process of finding and fixing software bugs can benefit from domain expertise — if a beneficial grouping or *clustering* can be found, each group can be handled as a unit by relevant developers. We propose a novel technique for grouping software defect reports. There are two goals when performing such a task: an accurate clustering (i.e., how similar are the clustered bugs?) and large clusters (i.e., how much time can be saved by parallelizing effort?). By measuring defining characteristics about bugs and comparing bugs across these sources of information, we produce a clustering technique that maximizes both of the aforementioned goals. We further validate the resulting clusters by showing that humans overwhelmingly agree that the associated defects are, in fact, highly related and could be triaged and potentially fixed in parallel. To explain by analogy, when maintaining a building, one might group related tasks to assign to the appropriate worker, to both expedite the task completion and to gain confidence that tasks will be carried out correctly. For instance, multiple different leaky pipes might all be assigned to a plumber while multiple wiring issues would be better handled by an electrician. In a physical building, it may be easy to distinguish a plumbing problem from a wiring problem. In complex software systems, however, such a grouping is not always obvious, and a new technique for assigning such groupings is a contribution of this thesis.

- **Efficient automated program repair** — Fixing bugs manually is costly and may even represent a losing strategy when trying to keep up with the ongoing stream of reported defects in real systems [46, p. 363]. Automatic program repair represents a chance to fix more of the many bugs reported in practice. In this thesis we present a novel approach to automated program repair that is more efficient, and more theoretically rigorous, than previous similar approaches. A popular program repair strategy is to apply small directed program changes to specific parts of the code, hoping to find a set of modifications that fixes the associated bug (e.g., [47]). We show that this naïve approach is sub-optimal when attempting to produce defect patches for real-world bugs. Notably, certain changes to a program are functionally equivalent by construction — to explain by revisiting the plumbing

analogy, checking the water pressure in each connected pipe segment on the same known-functional water line is likely redundant. In the case of program repair, by measuring functional program equality, we can avoid redundantly considering semantically equivalent patches. To validate potential program changes as an effective patch, traditional techniques use test cases to check for required program behavior. We further accelerate this process by prioritizing how we test patches, favoring a fail-early test ordering, based on adaptive learning from historical data. To revisit the building maintenance analogy, when trying to diagnose a faucet with no water flow, a naïve approach would be to start replacing *all* relevant pipes and fittings, hoping to find the leak. However, by examining blueprints and reasoning about the structure of the system, one might find related components that need not all be checked — if the plumber has found that water is reaching the *end* of a section of pipe segments, the intermediate segments need not be re-checked as they are demonstrably carrying water to the endpoint and thus are not the cause of the problem.

- **A human study of patch maintainability** — The lack of human-input throughout the automatic patch generation process makes the resulting bug fixes cheap, but may also degrade system quality over time. In a fully-automated scenario, lack of human oversight could lead to the creation of functionally-correct but potentially-unintuitive patches. To mitigate this concern we performed a human study measuring how well developers understand different types of patches. We show that with additional machine generated documentation, developers understand the resulting code as well as code patched by humans and, on average, in less time. This finding suggests that applying our patches continuously over time can reduce the cost of fixing software while maintaining system quality. Revisiting the building analogy a final time, imagine a building manager found a plumber that offers to work for *considerably* less money than previous plumbers. She may be concerned that the much cheaper work will be of lower quality. Intuitively, the cheaper plumber may be cutting corners and doing lackluster work to reduce costs, which poses challenges for future building maintenance.

The work presented in this dissertation reduces the cost of software maintenance while closely following the three important principles highlighted previously: generality, usability, and comprehensive evaluation. Clustering can be performed on any structured, automatically generated defect reports and we generate patches across various types of bugs across a range of system domains which strengthens the *generality* of the results in this thesis. We *comprehensively evaluate* the associated techniques by providing more traditional evaluations and results from two respective human studies, showing effectiveness as well as real-world applicability. Finally, the techniques described work "off-the-shelf" and require no additional human input or intervention, suggesting that they would be very *usable* in practice.

```java
/**
 * Opens the file pointed to by path
 * @param path the string representation of the file
 * @return the opened File object, or null if the file does not exist or path is null
 */
private static File openFile(String path){
   File openedFile=null;
   openedFile = new File(path);
   if(!openedFile.exists()){
      System.err.println("File does not exist:" + path);
      return null;
   }
   return openedFile;
}
```

Figure 1.1: A snippet of Java code to illustrate the presence and utility of natural language in code. Note the use of meaningful identifiers like `openedFile`, the phrase "does not exist" in the error string, and various related terms (e.g., "file," "open," "path," and "null") in the leading JavaDoc comment.

1.3 Scientific intuition — using latent information in software artifacts

Software development and maintenance processes result in a wealth of software artifacts, including direct deliverables like source code and documentation explaining the code's intentions but also less outwardly-visible sources of information such as repositories of fixed and unresolved bugs and databases of historical code change. Traditional approaches to improving software maintenance often favor narrow sources of information that allow for mathematically rigorous analyses, but cannot be applied to all artifacts [23, 24, 25, 26, 27, 28, 29, 30]. This means that a subject system must meet the strict information-specific assumptions of each tool for that tool to be applicable. We believe that there exists a wealth of oft-overlooked, more general information in software artifacts that can be helpful in improving the maintenance process in wider focused, more generally-applicable sense. The challenge then becomes implementing accurate analyses to extract and leverage such information.

One such piece of generic, but potentially instructive, information is the natural language inherent in code and other human-written software artifacts. For example, consider the natural language information in the Java snippet shown in Figure 1.1. The code itself, the string literals, and the associated comments all contain information related to the intent of the method. These language clues are non-functional in nature (i.e., systematically replacing all variable names in a program with randomly generated names yields a semantically identical, but likely less human-understandable, program). However, if we assume that developers write code to be correct (i.e., with good intentions, as people have done in the past [48, 49, 50]), we also expect the inherent language clues to be mostly accurate and explanatory. Historically, non-functional information, including natural language, has been underutilized when developing maintenance tools and analyses — only recently has the community begun to embrace such implicit data [51, 52]. For instance, a developer trying to find a bug associated with opening files might use the language in the code in Figure 1.1 as a guide to help find the defective statements. A maintenance tool could potentially leverage the same information to perform the same task,

and doing so automatically could save developers time when finding bugs. The work in this dissertation recognizes the utility of natural language and other sources of historically under-used information and exploits them to reduce the cost of software maintenance processes.

1.4 Metrics and criteria for success

The main goal of this dissertation is to reduce the cost of software maintenance overall by facilitating both bug finding and bug fixing. In Section 1.2 we outlined three overarching goals for this work: generality, usability, and concrete evaluation. This section both explains how we will measure performance in each of these areas and also outlines the criteria for success.

1.4.1 Generality

We focus on generality: a tool that fixes a broadly defined problem is likely to apply to more situations than one that is narrowly focused. Such applicability could translate into larger total cost savings, in practice, as the tool is able to save effort in more scenarios. Concerning generality, we now focus specifically on how it applies to finding and fixing bugs. With respect to finding bugs, we present an automatic defect report clustering technique (Chapter 3), that helps to assign related defects to the respective domain experts. We desire a clustering technique that is applicable to many defect finding tools and many different types of defects. To this end, we hypothesize that:

> **Hypothesis 1:** The defect report clustering tool presented in Chapter 3 can cluster a large class of defect report types produced by several applicable bug finding techniques (including multiple existing, available tools).

In Chapter 3 we show quantitative evidence to support this claim and also argue qualitatively that it is broadly applicable by extending the argument sketched above. We measure the number of bug-finding tool report types and programs that our approach successfully applies to, as well as noting any assumptions we make about the shape of the input. Our approach is successful quantitatively if it applies to multiple tool report types and programs and qualitatively if it makes no unnecessary assumptions about the shape of the input (i.e., if its design adheres to principled generality guidelines by construction).

Concerning bug fixing via efficient automatic patch generation (Chapter 4), we hypothesize that:

> **Hypothesis 2:** The patch generation technique in Chapter 4 is applicable generically to many types of bugs, by construction, and will produce patches for strictly more types of bugs than previous techniques.

We show quantitative evidence supporting this claim and also argue qualitatively that by construction, the technique does not assume the presence of a certain type of bug. We measure types of bugs using the established taxonomy of

Kaner, Falk, and Nguyen [53]. Success in this domain is requires that the technique can produce patches for at least as many bug types as GenProg and other competitive state-of-the-art program repair techniques.

1.4.2 Comprehensive Evaluation

In addition to traditional empirical correctness and cost concerns, we believe a comprehensive evaluation of a development or maintenance tool also requires evaluating human-centric notions of quality. This subsection discusses both of these complimentary evaluation concerns in turn. Success criteria is explicit in this section as the hypotheses are phrased in terms of quantifiable goals.

Evaluating software tools in real-world industrial scenarios using appropriate data sets and metrics has long been recognized as essential when developing tools with practical applications [54]. Perceived weaknesses of previous work in developing computing tools in general include the size, scope, and real-world application of the benchmark programs used to evaluate such techniques [55]. For instance, a comprehensive survey of mutation testing (a software maintenance concern for over 35 years) shows that the majority of data sets used to evaluate such techniques contain fewer than 100 lines of code [56, Table 9]. Modern systems are increasingly large and complex, often comprising thousands or even millions of lines of code (see Table 3.1 and Table 5.1 for examples) and thus small, limited evaluations may not generalize to real, state-of-the-art software. To avoid this common pitfall, we evaluate the techniques in this dissertation using thousands of defect reports and hundreds of real bugs from programs spanning several domains, containing millions of lines of code.

Empirical cost

The goal of this dissertation is to reduce maintenance costs and we are particularly interested in examining cost metrics. With respect to bug finding, the clustering technique presented in this thesis is designed to save humans time by allowing for similar defect reports to be handled in parallel. In this regard, we compute time savings in terms of the size of the clusters, which directly corresponds to the level of possible parallelization. We hypothesize that:

> **Hypothesis 3:** The clustering technique presented in Chapter 3 can automatically cluster over 50% of similar defect reports from large open source C and Java programs, comprising millions of lines of code, with few (less than 5%) false positives.

A false positive in this sense represents a defect report that is mistakenly clustered with dissimilar reports — in practice, false positives are detrimental to the goal of saving developer time and effort. The evidence in support of this hypothesis is discussed in Chapter 3.

Concerning bug fixing, we directly examine the dollar cost (a product of the computing time used) of automatically producing patches. We hypothesize that:

> **Hypothesis 4:** The efficient patch generation technique presented in Chapter 4 can further reduce the monetary cost of producing said patches by 50% when compared with the previous state of the art, while producing as many patches in practice.

Achieving such speedups over previous work transitively shows improvement over human-generated patches as well — we compare directly against previous tools which were significantly less costly than manual fixes [47].

Human-centric concerns

Automated software maintenance techniques are only effective in practice if humans perceive them to be of high reward and low risk; it is unlikely developers will use tools in which they see little overall value [45]. We specifically address these human-centric concerns in this dissertation to ensure a comprehensive evaluation of our proposed techniques. Examining the defect clustering technique, we hypothesize that:

> **Hypothesis 5:** Humans agree with the defect report clustering algorithm's notions of highly-similar reports presented in Chapter 3 (based on the perceived similarity of the underlying bugs) at least 90% of the time.

If humans deem the clustered defects to indeed be similar (i.e., the tool is producing what the humans want), it provides evidence that the tool would be useful and acceptable when responding to defect reports.

With respect to bug fixing, automatic patches save time [57], but lack the human intuition of manually written patches [58]. As such, there is some concern that applying them over time might negatively affect humans' ability to understand and thus maintain the code. We hypothesize that:

> **Hypothesis 6:** When augmented with automatic documentation in Chapter 5, the automatically generated patches presented in Chapter 4 will be as maintainable as those created by humans.

In this dissertation, one patch is deemed more maintainable than another if indicative questions about the first patch can be answered with the same accuracy, but require less thinking time, than those same questions about the second patch. Maintainability in this context is thus gauged by developers' accuracy when answering program understanding questions indicative of developers' questions in practice [59] about code containing different types of patches. We thus measure the time taken by human subjects as well as their accuracy when presented with patches and software maintenance questions. Confidence in the utility of our technique will be strengthened if the resulting automatically generated patches are as maintainable as those written by humans and cost significantly less. Chapter 5 directly evaluates this notion of human-centric patch quality with respect to future understanding and maintainability.

1.4.3 Usability

In this dissertation, we consider a technique usable if it requires minimal human input and works adequately "off-the-shelf," which encourages widespread adoption and thus has the potential to yields practical cost savings. We address these goals simultaneously for both bug finding and fixing. We hypothesize that:

> **Hypothesis 7:** The techniques outlined in Chapter 3 and Chapter 4 will require no additional human input and the results will be of *adequate* quality without any program or domain-specific tuning.

Adequacy of results is measured based on the metrics and success criteria described in Section 1.4.2. Notably, we provide evidence in Chapter 3 and Chapter 4 that the tools described in this dissertation are usable by making clear the lack of necessary domain-specific tuning while also providing a comprehensive evaluation, showing the efficacy of the tools in practice.

1.5 Broader Impact

The current state of corrective software maintenance shows an alarming trend: developers cannot handle the volume of bugs being reported in their software. Quoting directly from a Mozilla developer: "Everyday, almost 300 bugs appear [...] far too many for only the Mozilla programmers to handle" [46, p. 363]. Further evidence suggests that maintenance tasks, even when aided by automated tools, can be overwhelmingly expensive in practice. For instance, a global survey of the Google code base using FindBugs, a static analysis bug finder, yielded nearly 10,000 suspected defects [4]. Despite these warnings leading to 1,746 manual bug reports being filed, after a month of company-wide maintenance effort only 640 (37%) of the associated bugs had been fixed. This suggests that while automated techniques can be useful in practice, the human effort needed to conclusively solve the associated maintenance problems is still prohibitively expensive.

A similar longitudinal study was performed using the commercial Coverity static analysis bug finding tool [22]. They first note that in 2009, at the time of publication, 700 customers were using the tool on over a billion lines of code, which speaks to the breadth of such maintenance tools' deployment in practice. The authors further report a scenario where an anonymous customer ran the bug finder on a version of their product, finding 2,400 defects, only 1,200 (50%) of which were fixed over the development lifecycle of that version. Furthermore, when they reran the tool for the next version of the customer's product, it found 3,600 defects. This both echoes the sentiments of the FindBugs survey (i.e., that handling all available defect reports is difficult in practice) and further shows that corrective maintenance concerns are ongoing in nature (i.e., continue to evolve as the software does). In most development scenarios it is assumed that a product will continue to evolve to some degree. Current maintenance strategies thus represent a continuous battle; companies can only try to keep up with corrective maintenance concerns [46, p. 363].

The work outlined in this dissertation could have direct impact in both of the real-world scenarios presented above. First, the initial concern is that developers seem to be overwhelmed with the resulting defect reports when using static analysis tools. The use of an automatic defect clustering technique could allow for the removal of large groups of potentially spurious reports, thus allowing developers to easily narrow down and focus on those defects that represent the largest potential problems in the system. An effective, efficient automatic patch generation technique could then be used, for instance, to produce patches for lower-priority defects while developers focus on more complex bugs that might require significant human intuition and creativity. Additionally, users can have confidence that the techniques' output is of high quality, because of the human-based evaluations we present measuring cluster accuracy and patch quality. The examples provided in this section illustrate real-world scenarios where current maintenance strategies are inefficient (and thus not perfectly effective) and show how the techniques presented in this dissertation could practically reduce the overall costs of software maintenance, thus making a concrete impact in software quality.

1.6 Contributions and outline

The overarching thesis of this dissertation is:

> **Thesis:** it is possible to construct usable and general light-weight analyses using both latent and explicit information present in software artifacts to aid in the finding and fixing of bugs, thus reducing costs associated with software maintenance in concrete ways.

The primary contributions of this dissertation are:

- A technique for clustering automatically generated defect reports that outputs large, accurate clusters, suggesting considerable developer time savings (Chapter 3);
- A human survey showing that developers largely agree with the clustering of related defect reports produced by our technique (Chapter 3);
- An efficient automatic patch generation technique that minimizes both redundant patch checking and testing efforts (Chapter 4);
- A large-scale developer study showing that our automatically generated patches can be as maintainable as those produced by humans (Chapter 5).

Additionally, Chapter 2 outlines background and related work on software engineering and, more specifically, software maintenance. It describes the context of software maintenance as a part of the overall development process and gives additional information regarding the specific tasks associated with corrective software maintenance. This

chapter also outlines some of the associated challenges in software maintenance that represent potential cost reductions, which is a goal of this dissertation.

Chapter 2

Background and Related Work

"To know that we know what we know, and to know that we do not know what we do not know, that is true knowledge."

– Nicolaus Copernicus

THERE are ways to avoid inserting bugs during the software development process, but projects almost invariably ship with both known and unknown bugs [12]. *Software maintenance*, the focus of this dissertation, encompasses all developer activity after completing the initial system implementation [60]. These activities can include adding features as well as finding and fixing bugs; we focus primarily on bugs, or "corrective software maintenance". Corrective software maintenance can be further broken down into subtasks including, but not limited to: testing, formal verification, bug reporting, bug verification and triage, fault localization, bug fixing, code review, and continued software maintainability assurance. This dissertation specifically focuses on bug reporting and triage, bug fixing, and software maintainability. This chapter provides background information on the aforementioned corrective software maintenance tasks, with an emphasis on the problems that are specifically addressed by this work.

Section 2.1 highlights the problems associated with software bugs. Section 2.2 describes methods for avoiding bugs both during development and after deployment (i.e., as part of the software maintenance process). Next, Section 2.3 outlines ways that both humans and automated tools describe bugs. We characterize strategies for fixing such bugs in Section 2.4. Finally, Section 2.5 outlines issues associated with continued system quality in the face of maintenance activities.

2.1 Software bugs are prevalent, impactful, and expensive

Though the term *bug*, as it relates to computer science, has etymological origins related to actual insects [61], we generally abstract the term to mean "a defect that causes a reproducible or catastrophic malfunction." [6] This dissertation focuses specifically on *software bugs*, i.e., errors in the source code that produce undesirable behavior or results at run time. Officially, "bug," "defect," and "fault" may have nuanced meanings in certain contexts [6]. However, for the purposes of this dissertation, we use them interchangeably to generically mean errors in software, encompassing both the symptoms and the fundamental coding error that causes such symptoms. Finding and fixing bugs requires examining both of the aforementioned concepts and thus we do not distinguish between the two in this document.

Bugs are prevalent in software both before and after deployment [12]. For example, if we examine the Mozilla suite of browser and web-based programs, as of March 1, 2014, they report 309,736 bugs over the 17 year documented lifetime of their product suite [62]. While 114,958 of these have been "resolved" (i.e., fixed or otherwise deemed non-detrimental [15]), a staggering 194,778 (63%) remain unresolved (i.e., unexamined or unfixed). This trend is not unique to Mozilla — consider Figure 2.1 which exhibits a similar trend for the OpenOffice suite of professional productivity software, showing both the rate of incoming bug reports and the rate of resolved bugs over the course of 12 years. While Mozilla and OpenOffice may not be indicative of all software products, these figures show that even large, popular systems can be rife with defects.

Having established that bugs are pervasive in modern systems, we next examine their impact in the real world. Chapter 1 mentions the effects of both the infamous "Y2K bug" and the "Zune bug". To summarize, it is estimated that the former defect elicited additional, non-trivial maintenance costs for up to 75% of all software world-wide and presumably could have caused system-wide ambiguities, had it not been addressed in the affected systems [17]. The "Zune bug" caused complete system unresponsiveness for an entire day for many Microsoft Zune owners and amounted to a single-line error. As another example of the potential impact of software bugs, in early 2014 Apple found a security bug in their operating system code that had allegedly existed for around 18 months [63,64]. The bug in question affected the encryption protocols, making it security-critical, for desktop, laptop, and mobile users alike, which translates to a wide human impact. These examples show that bugs can have dire consequences to end users and thus represent a serious problem for software developers [65].

Another way to examine the impact of bugs is through monetary cost. Software defects account for a considerable portion of the cost of modern systems, both in terms of lost computing power as well as the expense associated with actually generating patches to fix the associated defects. As mentioned in Chapter 1, it is estimated that the global cost of defects is as much as £192 billion annually. While manual, human-written fixes have historically been the most popular method for addressing defects, paying humans with the requisite expertise to craft patches is costly in practice. Corrective software maintenance is so expensive, in fact, that companies often struggle to keep up with the multitudes

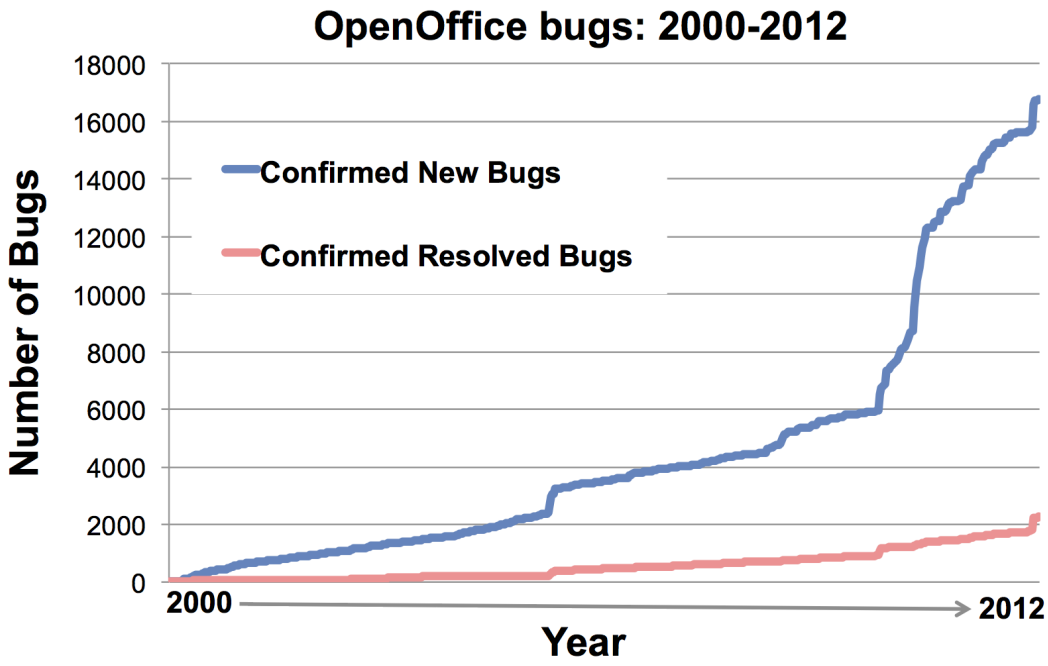

Figure 2.1: The graph plots the number of reported bugs against the number of bugs that were resolved for the Apache OpenOffice project from 2000 until 2012. Note that the rate of incoming bugs far exceeds that of resolved bugs [66].

of newly reported bugs [46, p. 363].

To cope with the ever-increasing deficit between incoming and resolved bugs (referring, for instance, back to Figure 2.1), companies have recently gone so far as to offer "bug bounties" to outside developers. In these bug bounty programs, companies like Mozilla and Google bid against one another (in excess of US$3,000 in some cases) to attract outside developers to find and fix crucial bugs.[1] Microsoft recently set a new high bar for bug bounties, offering up to US$100,000 for discovering novel security vulnerabilities in the Windows operating system [67]. The recent introduction of such steep bug bounties suggests that companies are aware of the widening gap between the number of issues users experience in the field and those that are eventually fixed. Companies seem eager to find solutions to both finding and fixing bugs, which are two of the main issues addressed in this dissertation. This new trend demonstrates that spending even large amounts of money is economically feasible when compared with in-house maintenance, making low-cost tools a viable alternative. Additionally, the fact that companies are wiling to outsource maintenance tasks to unaffiliated developers suggests that they would be willing to try other, perhaps automated, non-traditional software maintenance methods.

[1] http://www.computerworld.com/s/article/9179538/Google_calls_raises_Mozilla_s_bug_bounty_for_Chrome_flaws

2.2 Common strategies for avoiding bugs

Because of the significant impact and cost of bugs, avoiding them is of paramount importance during both development and subsequent system maintenance. We first consider the possible stages of software development. The waterfall model, a traditional, simplified approach to developing a piece of software, outlines a well-defined flow and order for development activities [68]. One might first gather *requirements* for what behavior or functionality is desired for the system. The process then continues with high level architectural system *design* and *specification* gathering which leads to the concrete *implementation*. Once the code base has been established, *testing* and *verification* of the system can occur and then the software is generally deployed. After deployment, *maintenance* is generally performed to add or correct functionality. The various techniques described in this section are organized temporally according to this simplified model, for the sake of presentation, though they could potentially apply to various parts of the software development process.

We describe techniques for avoiding bugs through the lens of information use, as the types of information used and the underlying techniques for extracting and analyzing such data often apply to the techniques presented in this dissertation.

2.2.1 Avoiding bugs during design and implementation

Developers generally write correct or close-to-correct programs — bugs are intentionally avoided, if possible, when writing code [48, 49, 50]. Despite this theory, dubbed the "competent programmer hypothesis," bugs are a fundamental hindrance in modern software systems [12]. There are several techniques for avoiding bugs throughout the development process — one loose classification of the various approaches distinguishes best practices and metrics from concrete tools. Abstractly, there are many coding *anti-patterns* that, when avoided in practice, can increase system quality [69]. Examples of detrimental design anti-patterns include "spaghetti code" referring to overly tangled control structure; "lava flow" where older code "solidifies" and is neglected; and "cut and paste coding" or *code clones* wherein code is copied into many different contexts rather than extracted and called uniformly in a centralized nature [70]. All three anti-patterns can largely be characterized as the degradation of program structure during software evolution. Program structure is a key source of information for all three thrusts of this dissertation (see section Section 1.2).

Code quality metrics are a complimentary abstract method for measuring and promoting known beneficial code characteristics and thus ideally avoiding bugs. Examples include code *readability* [71], a human judgment of how easy a text is to understand [72]; *cyclomatic complexity* [73] which measures the shape and connectedness of the structure of a program; as well as *coupling and cohesion* [74] which together measure the degree to which program modules rely on one another and belong together, respectively. These metrics largely use both natural language (e.g., as illustrated in

Figure 1.1) and program structure, which are two pieces of program information that underly the techniques in this dissertation.

There are also a number of *concrete tools* at developers' disposal to facilitate writing code to avoid bugs. Program *search tools* use natural language similarity to concretely identify abstract concepts in code — for instance, to avoid code clones by duplicating existing functionality [75]. Code *navigation and visualization tools* offer developers high-level views of the structure of the software at various granularities (e.g., [76]). Such information can be instrumental when trying to implement and enforce good system design and structure, which can help to avoid anti-patterns. Finally, *automatic refactoring tools* can help to apply predefined program transformations to code to actively avoid system quality degradation [77]. Refactoring tools exploit recognizable code patterns and program structure to then perform known template-based code changes.

All of the design and development-based analyses and techniques for avoiding bugs described in this section leverage information such as natural language and program structure, which will also be instrumental in our post-development (i.e. maintenance-based) approaches to finding and fixing latent bugs. The utility of such information in preliminary stages of the software development process suggests it may be useful in later stages as well. We explore this intuition in Chapter 3, Chapter 4, and Chapter 5.

2.2.2 Avoiding bugs before deployment

There are additional approaches to find bugs once a system's implementation has been mostly completed but before it is deployed — this section describes several examples of such techniques. Many of these techniques could easily be applied during development or as part of the software maintenance process as well, but we will discuss them in terms of pre-deployment strategies for the sake of simplicity.

Testing

Software testing has long been a crucial technique for exposing bugs in programs and continues to progress as an active area of research [78]. Testing is a dominant part of software development and maintenance, sometimes accounting for 50% of the overall lifecycle of a project [79]. At a high level, software testing involves finding inputs to exercise some part of a system, then comparing the resulting programmatic output with predetermined, expected output [8]. Testing software allows developers to gain confidence that their existing implementation both adheres to the program's particular specification (e.g., performs necessary tasks) and also follows implicit language specifications (e.g., contains no null pointer dereferences). Creating a set of test cases (called a "test suite") to exercise the important functionality in a given program is often very difficult, which leads to inadequate test suites in practice [19]. Testing has traditionally been a manual, human-centric task [78], as such, testing can often be very costly to perform both comprehensively and

often [19, 79]. Automated test generation or prioritization techniques have been developed to mitigate some of this cost (e.g., [80]). The goal of test generation is to automatically cover untested statements or branches in a program by specifically directing execution to the desired statements [80]. Test suite prioritization aims to reduce the runtime cost of testing, for instance, by favoring tests likely to fail first [81]. These techniques have proven successful at reducing the cost of creating and running tests but often lack the expressive power to generate widely focused, adequate test suites quickly in all cases [78].

Because testing is such an important and dominant part of the software development process [79], we desire *adequate* test suites (i.e., those that cover as much of a program's code and behavior as possible). Increasing the adequacy of a test suite helps to ensure overall system quality. There are several ways to measure test suite adequacy including coverage [8] (the goal of which is to exercise as many statements or branches as possible) and mutation testing [56] (which measures how many program changes, which are used as a proxy for possible bugs, are caught by a given test suite). We describe mutation testing, as it relates directly to work presented in Chapter 4, in more detail. This notion of test suite adequacy takes into account the number of defects a given suite can expose. Intuitively, eliminating bugs during the testing phase of software development helps to increase confidence that there will be fewer defects after deployment. While testing can help to expose *unknown* bugs, one can purposely seed a program with *known* bugs and measure the fraction of those seeded bugs a given test suite uncovers as a way of approximating adequacy. Mutation testing follows this methodology (i.e., seeding defects in a particular manner using code changes from predefined templates) by automatically applying different mutations (to create mutants) as an attempt to introduce faulty behavior that may mimic real bugs [56, 82]. Mutation operators commonly mimic the types of errors a developer might make [83]. Untested parts of a system are exposed if the mutations change the program behavior in a meaningful way and the test suite fails to detect those changes; this suggests the test suite is inadequate. Testing is expensive in practice and thus we desire adequate test suites to ensure the cost incurred is spent productively.

There are several critical challenges for mutation testing that closely mimic challenges addressed by our research discussed in detail in Chapter 4 and Chapter 5. One such challenge is the prohibitively large search space of possible program mutations. Even moderately-sized programs, when combined with a few mutation operators, can result in impractically large search spaces when compared to the size of the original program [56]. Further, combining multiple mutations makes testing *all* possible mutants generally infeasible [56]. A second challenge is the *high cost of testing* — in the context of mutation testing, the hope is that a given mutant will fail one test, which might require executing the entire suite in practice. Finally, one of the largest challenges associated with mutation testing is the difficulty in determining if a mutant actually changes program behavior (and thus should be caught by tests) or is equivalent to the original (and thus should not be) [84]. This concept is called the *equivalent mutant problem* and is an active area of research [56]. Early work in mutation testing attempted to solve this problem by sampling mutants (e.g., [85, 86]) or systematically selecting which mutations to make (e.g., [87]). Previous work in detecting equivalent mutants considered

many possible approaches: using compiler optimizations [88, 89], constraint solving [90], program slicing [91, 92], attempting to diversify mutants via program evolution [93], and code coverage [94]. In Chapter 4 we discuss conceptual similarities between the three problems described here, as they relate to mutation testing, and a program equivalence optimization used to speed up our patch generation technique.

Static Analysis

The practice of analyzing programs statically to find known faulty code patterns has grown in popularity and helps to mitigate the cost of traditional bug finding techniques such as testing [12, 22, 95, 96, 97, 98]. Static analysis involves reasoning about how a program might behave at runtime, without actually having to execute it, to establish properties about the program (e.g., correctness). For example, dereferencing a null pointer can lead to erroneous runtime behavior and thus we desire a method for recognizing the corresponding faulty code patterns in practice. Statically reasoning about all programs' runtime behavior is undecidable, but in many cases we can identify certain behavioral properties (e.g., potential defects) based on language semantics and program structure. The overhead of having to actually execute a program is avoided by reasoning about the static code, which helps to reduce the cost of finding bugs. Additionally, test suites are often specific to a given program while static analyses can be written once and applied to many programs generically [99, 100].

Static code analyses used to find bugs do, however, have several drawbacks. It has been proven that there is no perfectly accurate method for statically measuring non-trivial properties about generic programs [101] and for this reason static analyses suffer from false positives (e.g., code may be reported as "faulty" when it is in fact functionally correct), false negatives (e.g., actual bugs are not recognized), or both. Additionally, whereas for software testing the main challenge is encoding the specification in terms of program inputs, expected outputs, and comparators, a prominent challenge associated with static analysis is algorithmically reasoning about the runtime behavior of a program on all possible inputs and, in particular, encoding a machine specification of "correct" behavior [102]. Failure to recognize a useful input in testing can translate into un-exercised parts of a program which can lead to missed bugs. Similarly, failure to recognize or properly specify a faulty code pattern can result in incorrectly reasoning about that part of the specification when using static analysis. If the effects and costs of these obstacles can be minimized, static analyses can be effective at finding many, but generally not all, bugs in practice [4, 22].

Formal Verification

In certain situations, developers might desire more certainty about the correctness of their programs than testing or static analysis bug finders can provide in practice. Software for automobiles, airplanes, and medical devices, for instance, have safety-critical concerns that require the utmost amount of care with respect to correctness. In this

scenario, one might *formally verify* certain properties of a program in order to obtain complete certainty about aspects of correctness. Techniques have been developed to automatically check whether a program adheres to a pre-defined specification [103, 104, 105, 106]. Formal verification (one example of a larger classification of formal methods) requires a specification of the desired program functionality and a logical model of the underlying system. One first constructs a logical formula from the program model and specification such that the program is correct if and only if the formula is true. The problem then becomes proving or disproving the formula which can be performed manually or via a theorem prover. While manual proof techniques can be extremely cumbersome, automatic theorem provers often fail to return an answer. Another weakness of formal verification is that it requires a model of the functionality of only the underlying program, which can abstract away low-level details such as memory safety (e.g., null pointer dereferences). Building such models is also historically difficult for humans which presents scalability concerns as program size increases [107]. Another weakness of such approaches is the need for complete formal correctness specifications for all interesting properties of a program which are difficult to develop and thus less common in practice [108, 109]. Finally, the tools used to perform rigorous verification, like theorem provers (e.g., [110]), are often cumbersome and can be difficult to use and understand [111, p. 11]. As a result of these weaknesses, formal verification often does not scale well in practice [112], though recent work has shown promise in this area by using automated specification synthesis [113]. While formal verification is often too heavyweight to find a wide range of bugs in industrial-sized systems, recent work has shown small-scale, incremental approaches to be tractable in practice [114]. Applications that have explicit specifications have been successfully verified as well, although such artifacts are rare in practice [115].

Summary of pre-deployment bug avoidance techniques

Although the techniques described in this section are all effective at their respective bug finding tasks, software still ships with bugs in practice [12]. As such, it is necessary to perform corrective maintenance to find and fix such bugs after deployment. There are many ways to expose bugs throughout the software development process, but several thematic weaknesses underly many of these techniques. One such weakness is the overall cost associated with finding bugs. An additional concern with existing bug finding and avoidance techniques is that they often lack usability. Many existing processes require human intervention or scarce development artifacts. Comparatively, the techniques presented in Chapter 3 and Chapter 4 require only prevalent software artifacts and no additional human intervention. Chapter 5 further investigates the usability of our automatically generated patches.

2.3 Bug reporting as a means to describe software defects

Despite the techniques described in Section 2.2.2, bugs exist in deployed software [12]. Bug reports are one structured method for describing defects exposed after deployment, ideally giving developers as much information as possible

to facilitate a patch. Bug reporting is performed both manually, by users, and automatically using maintenance tools. These two strategies generally offer developers slightly different types of information when trying to find and fix the associated bugs, but both suffer from a common problem: duplicate reports. To understand how duplicate reports might arise, we must first examine the lifecycle of a bug. Bug reports generally follow a structured workflow, starting out as "unconfirmed." Once a developer has *verified* a bug by reproducing it, she triages the bug by assigning it a *severity* or *priority* and assigning it to a developer with appropriate domain knowledge. Bug reports remain open until they are fully addressed, which then renders them *resolved*. Resolved bugs assume one of many final subclassifications, including *duplicate, invalid, fixed, wont-fix,* or *works-for-me*, that explain how and why they were resolved [15]. Regardless of whether a bug is reported manually or automatically, it can suffer from duplication. In practice, duplicate defect reports complicate aggregating all of the available information about a given bug which confuses the process of finding and fixing said defect. We investigate both bug reporting methodologies in this section, in the context of duplication.

2.3.1 Manual bug reporting

Allowing end users to report bugs found during program execution is considered an effective way to expose software bugs that were not previously caught via traditional means (e.g., testing or static analysis — see Section 2.2) [116]. Generally, bug reporting systems require users to enter a title and description of the symptoms experienced in the form of natural language text. Optionally, a stack trace or error message can be included to further describe the defect, but as few as 11% of manual defect reports contain these types of "technical" information in practice [46]. This reporting framework can be effective at exposing software bugs [117], even when bug reports suffer from low-quality or missing information [15].

In practice, many users might experience the same bugs, but describe the associated problems differently due to differing contexts or lack of domain knowledge. This phenomenon manifests as duplicate defect reports, which have long been recognized as an important issue in software engineering (e.g., [5, 118]). Automatic techniques have been developed to eliminate duplicated human-created bug reports, thus saving developers effort in critical parts of the maintenance process [119, 120, 121].

2.3.2 Automatic bug reporting

Waiting for users to encounter faults in the field and report them manually may be undesirable because it can result in high cost to said users and potentially low-information bug reports. The existence of user-submitted bug reports assumes that the users in question must have actually experienced the associated problem, which can negatively effect users' perception of a piece of software. Additionally, manual bug reporting represents time and effort a user has to expend, which is an additional cost to consider. Many successful static bug finding tools have been developed in

response to these concerns [22, 97, 98, 104]. These tools generally try to find known-faulty code patterns in systems to expose likely defects (see Section 2.2.2 for more detail).

Duplicate automatically generated bug reports pose the same problems as their duplicate manually reported counterparts — notably, they complicate the process of finding and fixing the underlying defects. This represents an additional burden, and thus increase in cost, for the bug triage, fixing and tracking processes. Detecting duplicate automatically generated defect reports is a relatively unexplored area — Chapter 3 describes one such technique. We are aware of only one other approach that addresses this issue. To expose and leverage groups of similar reports, Kremenek *et al.* proposed a clustering technique for automatically created defect reports [38]. Their technique exploits code locality to cluster related defects with the goal of improving severity rankings and ultimately reducing false positive reports.

2.4 Fixing bugs, both manually and automatically

The general goal of finding bugs is to eventually fix them. Like many of the software development and maintenance tasks discussed previously, the difficult process of bug fixing has classically been carried out manually, but recent techniques in automated program repair show promise in reducing the human burden associated with these tasks. This section describes the general practice of debugging and details a state-of-the-art program repair technique.

2.4.1 Manual bug fixing

Software bugs are cases where a program's implementation fails to meet its specification. To quote the Board of International Software Testing Qualifications directly:

> "A human being can make an error (mistake), which produces a defect (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it shouldn't), causing a failure." [7, p. 11]

At a high level, repairing a bug consists of bringing the implementation more in line with its specification such that the symptoms of the bug are no longer exhibited. Important first steps in this process include: fully understanding the program and defect in question — that is, something about the current program implementation or specification is flawed and the developer is tasked with discovering how it should be changed to encode the desired, correct behavior. This can often be complicated by limited or missing information about the bug in question [122]. For example, stack traces can help locate the exact spot of a failure in a piece of code, but as few as 11% of human-created defect reports contain them in practice [15]. Once the desired behavior is clear, bugs are typically "localized" — i.e., the code that is causing the undesirable behavior has to be identified. Finally, the developer must then change the faulty code to elicit the desired behavioral changes. A high-quality bug fix should correctly rectify the faulty behavior while leaving

unrelated, functionally-correct behavior intact [65]; developers have historically favored smaller patches because their correctness is easier to verify and thus they may represent smaller future maintenance costs [123].

The goal of debugging is "to identify the [related] chain [of program statements], to find its root cause (the defect), and to remove the defect such that the failure no longer occurs." [124] A developer attempting to fix a bug presumably has at least the basic information associated with the symptoms of the failure, potentially gathered from a bug report. There are several tools designed to leverage such information in order to facilitate the debugging process. The earliest debugging tools allowed the developer to manually step through program statements at runtime, noting the path of execution and the values of relevant variables to understand the program behavior [125]. Further comparative information can be gained by examining faulty execution paths in the context of correct program executions to further narrow down the set of possible faulty statements [37, 32, 42] — this is often called fault localization. By not requiring a human to monitor the faulty program execution, these tools further automate the bug finding process. Additional progress was made by concretely identifying faulty statements based on examining how sets of program changes alter program behavior [126]. As a final example of the breadth of debugging tools, we highlight Valgrind. As a debugging tool, it is focused more narrowly on traditionally-difficult defects, for instance, by performing specific runtime monitoring to debug memory errors (which may be relatively opaque given traditional debuggers) [127].

Manually removing bugs can be difficult because it requires developers to fully understand many different parts of a potentially-large code base and solve diverse problems (e.g., race conditions vs. null pointer dereference) on a case-by-case basis. Program understanding is both central to finding bugs and also a very broad, ill-defined problem. Strategies for understanding programs can vary widely depending on the associated task or bug, code base, developer, time frame, and domain [128] — see Sillito *et al.'s* survey of developer questions and corresponding information needs for more information [59]. Because of both the difficulty associated with manually fixing bugs and also the high volume of defects in modern systems [5], the cost of bug fixing is extremely high (see Section 2.1).

2.4.2 Automatic bug fixing

Debugging comprises several difficult and time consuming tasks including characterizing, locating, and fixing defects. Reducing the cost of any single part of this process represents a significant reduction in the overall cost of software maintenance. Consequently, this section focuses specifically on techniques that facilitate bug fixing. In response to the cost of traditional, manual bug fixing, automated program repair has gained popularity since 2009 [39, 129]. Techniques for automatically effecting patches for known defects employ a wide range of approaches — many of the state-of-the-art tools are discussed in this section. To produce a patch for a given bug, these tools need, at a minimum, knowledge of both the buggy implementation (i.e., what the problem is) and the correct specification (i.e., what the

desired behavior is). A program's source code or binary often suffices for the implementation description. Specifications can be practically represented as test suites [129] or, more formally, as invariants or contracts [130].

GenProg — GenProg, a program repair technique based on genetic algorithms, is a focus in Chapter 4 and Chapter 5 of this dissertation. At a high level, GenProg systematically generates potential patches by making small sets of directed code changes. It then validates each potential patch in turn by checking to see if that patch both fixes the defect in question and also retains all desired program behavior [47, 129]. This strategy exemplifies so-called *"generate-and-validate"* program repair techniques [131]. A "patch" in this sense is a collection of atomic program changes (e.g., *additions*, *deletions*, or *swaps* of existing program statements). To favor altering likely-defective parts of the program, GenProg employs existing fault localization techniques (e.g., [32]) to probabilistically select the sites for potential code changes. GenProg assumes the necessary statements for a patch exist elsewhere in the program as a matter of practicality [48], and thus only manipulates existing statements rather than creating new code. To choose which program changes to make, GenProg adapts biological principles to simulate the evolutionary processes found in nature. Examples include small chromosomal shifts that may occur naturally as well as large, sweeping combinations of changes that might represent breeding in the wild [132]. To validate candidate patches, GenProg assumes the existence of an adequate regression test suite (i.e., a set of test cases that encodes necessary program behavior) and one or more test cases describing the associated bug. We define a *valid* patch to be one that passes all regression tests as well as all test cases encoding the buggy behavior. Finally, as automated program repair is intended as a supplement to traditional, manual bug fixing, GenProg is evaluated by comparing the monetary cost of the required computation time with the cost of paying a human to fix the same bugs [47]. This dissertation focuses on GenProg for two main reasons: the technique's implementation is freely available and also works out-of-the-box; and its evaluation framework is both competitively large [47] and also available to make direct cost and performance comparisons.

Repairing general software defects — While the work in this dissertation focuses on GenProg, many comparable automated program repair techniques have been developed to target general software defects. Arcuri proposed using genetic programming (GP) to repair programs [133]; and several authors explore evolutionary improvements [134] and bytecode evolution [135]. ClearView notes errors at runtime and creates binary repairs that rectify erroneous runtime conditions [39]. The ARMOR tool replaces library calls with functionally equivalent statements: These differing implementations support recovery from erroneous runtime behavior [136]. AutoFix-E builds semantically sound patches using software testing as well as Eiffel contracts [130]. SemFix uses symbolic execution to identify faulty program constraints from tests and builds repairs from relevant variables and constructive operators to alter the state of the program at the fault location [137]. Kim *et al.* introduced PAR, which systematically applies mined bug repair patterns from human-created patches to known faults, leveraging semantic similarities between bugs and human expertise [58]. Debroy and Wong [138] use fault localization and mutation to find repairs. The recent success of these generic program repair techniques has made automatic patch generation a popular area of research [139].

Repairing specific classes of defects — Several other repair methods target particular classes of bugs. AFix generates correct fixes for single-variable atomicity violations [26]. Jolt detects and recovers from infinite loops at runtime [36]. Smirnov *et al.* insert memory overflow detection into programs, exposing faulty traces from which they generate proposed patches [27]. Sidiroglou and Keromytis use intrusion detection to build patches for vulnerable memory allocations [28]. Demsky *et al.* repair inconsistent data structures at runtime via constraint solving and formal specifications [29]. Coker and Hafiz address unsafe integer use in C by identifying faulty patterns and applying template-style code transformations with respect to type and operator safety to correct erroneous runtime behavior [30]. While these techniques are generally effective within their respective domains, they lack generality. The specific classes of bugs targeted by such techniques (e.g., atomicity violations and unsafe integer use) are generally disjoint and thus a large tool suite would be required to address all relevant concerns in practice.

Automated program repair is a fast-growing and exciting area, as evidenced by the multitude of recent techniques to produce various types of defect patches. We believe that three key challenges remain: cost, generality, and usability (see Chapter 1) — this dissertation will address all three issues in turn. Chapter 4 provides significantly more detail on our baseline approach, GenProg, as well as several fundamental improvements that reduced the cost of automatically creating patches for a range of defect types. Chapter 5 studies the future maintainability of the resulting patches, a usability concern that few other techniques have historically taken into consideration.

2.5 Ensuring continued system quality throughout the maintenance process

Whether performed automatically or manually, system debugging necessarily introduces code changes. As discussed in Section 2.4.1, a key component of debugging is understanding both the existing program and the associated problem. Code changes may affect the future understandability of a software system, especially if performed in high volume over a period of time. This section discusses the issues related to continued software quality throughout the maintenance process.

2.5.1 Software maintainability and understanding

Software evolution can be defined as the "sequence of changes to a software system over its lifetime; it encompasses both development and maintenance." [140] Software evolution can naturally increase system complexity while simultaneously decreasing system quality if care is not taken to prevent such deterioration [21]. According to federal standards, *maintainability* is "the ease with which maintenance of a functional [software] unit can be performed in accordance with prescribed requirements." [141] Intuitively, if a system grows more complex and code quality decreases, it may also be harder to maintain. One of the main concerns associated with maintainability is "understandability" — loosely defined as the ease with which humans are able to comprehend the meaning or underlying functionality of a piece of

code — which is recognized as a difficult and time-consuming part of the maintenance process [142, 143]. Program understanding is also a crucial part of finding and fixing bugs [59] and thus represents a serious concern with respect to the software maintenance process. Aggarwal *et al.* argue that being able to understand and reason about code is central to the concept of maintainability [144].

Having motivated *maintainability* as an important abstract concept we desire a way to concretely measure it in practice. We first describe a number of historical, limited approaches. Welker *et al.* describe a single metric, the Maintainability Index, to statically determine the maintainability of source code [145]. This metric takes into account a number of other software quality metrics, including Halstead's program volume [146], McCabe's cyclomatic complexity [73], and average lines of code. In subsequent work, Heitlager *et al.* presented several criticisms of the original Maintainability Index and suggested potential improvements such as mapping system-wide characteristics of maintainability to source code properties and determining appropriate measurements for each of these properties [147]. Kozlov *et al.* attempt to correlate various software metrics with the Maintainability Index described previously and find that no single analysis can definitely describe the relationships between maintainability and established software quality metrics, suggesting the Maintainability Index may be an imperfect gauge of maintainability in practice [148]. The idea that there is "no silver bullet" is echoed in the work of Riaz *et al.* [149] and Nishizono *et al.* [150].

Unlike this historical work, we do not measure maintainability through software metrics designed to model human effort and cost. Instead, we gather an objective measure of understanding directly from humans and attempt to elucidate maintainability *a posteriori*. In Chapter 5 we investigate the potential consequences associated with applying machine generated patches (see Section 2.4.2) to a system by directly measuring the maintainability of several types of bug fixes by humans, rather than using historically-suspect metric-based approaches. We use program understanding questions that directly mimic those that developers ask when performing maintenance tasks and trying to comprehend code in the real world [59] to more objectively gauge maintainability. Since we require only a method for evaluating maintainability (i.e., rather than a prescriptive model) we can do so directly using real-world understanding queries that may be more objective and thus more indicative of concrete system maintainability.

2.5.2 Documentation

One approach to program maintainability and, specifically, program understanding is to document code to explain *what* the code is doing, *how* it is being accomplished, and perhaps *why* it is happening [151]. This supplementary information can help future developers comprehend existing code. Documentation has long been recognized as crucial to program understanding and thus software maintenance [152, 153, 154]. In a 2005 survey, NASA noted that a significant barrier to code understanding and reuse is poor documentation [155]. Documentation can be either external to the code (e.g., in

the form of architectural diagrams, system manuals, or code change commit messages) or internal (e.g., at the API boundary or in-line to explain program functionality).

There have been a number of proposed approaches for automatically documenting particular software aspects (e.g., exceptions [156] or API usage rules [157]). In this dissertation we augment patches with a slightly modified version of the DeltaDoc tool [151]. DeltaDoc is a tool for reasoning statically about code changes and synthesizing natural language explanations of the concrete changes that may affect a program's runtime behavior. DeltaDoc uses static code analysis which is described in more detail in Section 2.2.2. Synthesizing human-readable explanations of code changes requires translating logical formulae or relevant variables into English-language templates based on the structure of the underlying code and change.

In Chapter 5 we explore the effects of DeltaDoc's machine generated documentation on the understandability of automatically generated patches to show evidence of their efficacy in practice. Notably, we hypothesize that because machine generated patches do not benefit from human intuition it is less clear how to answer the question of "why" a code change is made, but that supplementing the change with information about "what" the patch does may help to aid future understanding.

2.6 Summary

This chapter presents background information and related work associated with finding and fixing bugs in real-world software. Section 2.1 explains the prevalence, impact and cost of software bugs. Strategies for avoiding bugs before product deployment are described in Section 2.2. Bug reporting, both manual and automatic, is outlined in Section 2.3. Section 2.4 describes strategies for facilitating bug fixing, both in terms of manual, developer-based patches and those created by machines. Finally, Section 2.5 discusses maintainability and documentation in the context of ongoing system quality. Subsequent chapters build upon this existing work to facilitate the software maintenance process, reducing cost by providing *general* and *usable* maintenance techniques while admitting *comprehensive evaluations*.

Chapter 3

Clustering Static Analysis Defect Reports to Reduce Triage and Bug Fixing Costs

"There has never been an unexpectedly short debugging period in the history of computers."

– Steven Levy [158]

3.1 Introduction

A critical part of the software maintenance process is identifying and addressing unknown defects. While defects can be reported by end-users or found during testing, such approaches are expensive and can typically only reveal bugs exercised during execution. In response to these limitations, many automatic bug finding techniques have been developed (e.g., [12, 22, 97, 98, 104, 127]).

However, false positives, spurious defect warnings, and duplicate defect reports can negate the potential time savings of such bug finding tools [159, 160]. While existing approaches have focused on reducing false positives and spurious warnings, the problem of duplicate defect reports *produced by automated tools* has not been investigated (cf. [119, 120, 121]). Such duplicate reports are an increasing problem in industrial practice: static analysis defect finders commonly require program-specific tuning to eliminate large groups of spurious reports. Even when programming patterns are ignored to reduce false reports, such tools can still produce groups of highly-related defects. We found that over 30% of the automatically produced defects examined in this chapter were duplicates, suggesting that a technique capable of producing clusters of highly-related automatically produced defect reports could save a considerable amount of developer effort.

In this chapter, we explore an automatic technique that clusters duplicate or related defect reports, increasing the utility of existing bug-finding tools. In this context, "related reports" are those that developers can triage at once, and ideally that all admit a similar fix; clustering related reports can thus save maintenance effort by allowing for handling similar reports in parallel. We believe that tool-generated defect reports can be clustered automatically because duplicate reports often share syntactic and structural similarities. Although some similarities can be syntactic, critically, not all duplicate defect reports involve syntactically-identical code that is easily identifiable as such [48]. While tools for detecting both syntactic similarity in code [161, 162, 163] and conceptual similarity in natural language descriptions of defects [119, 120, 121] are established, tools for finding syntactically different, semantically similar automatically generated defect reports are not.

We therefore propose a parametric technique to cluster defect reports using similarity metrics that leverage both syntactic and structural information produced by static bug finding tools. The technique takes as input a set of defect reports produced by a static bug finder and partitions it into clusters of related reports. The produced clusters must be accurate in order for the technique to be useful, because misclassification negates some portion of the provided time savings. We thus favor a clustering that maximizes the size of clusters produced (i.e., saves time) while ensuring that the clustered defect reports are in fact similar (i.e., is accurate). Because cluster size and accuracy are conflicting goals, our algorithm can be adjusted to favor one or the other as desired, producing a Pareto-optimal frontier of options.

We evaluate our technique on 8,948 defect reports produced by two popular static analysis techniques (Coverity [22], a commercial tool, and FindBugs [98], an open source tool) spanning eleven benchmarks totaling over 14 million lines of code in multiple languages. Since we are unaware of any previous techniques specifically designed to output clusters of defect reports produced by static analyses, we use existing code clone detection techniques as baselines. We find that our technique consistently clusters defect reports accurately.

The main contributions of this chapter are as follows:

- We propose a lightweight, language-independent technique for clustering defect reports produced by existing state-of-the-art static defect detectors.

- We empirically compare our technique against code clone detection tools using defect reports from both Coverity's Static Analyzer and FindBugs on large programs written in both C and Java. Our tool is capable of larger reductions in the overall set of defect reports when cluster accuracy is required.

- To ground our technique, we present a human survey in which participants overwhelmingly agree (99% of the time) that clusters produced by our technique in fact contain reports that should be triaged together.

Defect Report A:
File:
`/drivers/isdn/i4l/isdn_ppp.c`
Suspect Variable:
`lp->ppp_slot`

```
1  printk(KERN_DEBUG "Receive CCP
2    frame from peer slot(%d)",
3    lp->ppp_slot);
4  if (lp->ppp_slot < 0 ||
5      lp->ppp_slot > ISDN_MAX) {
6    printk(KERN_ERR "%s:
7      lp->ppp_slot (%d) out of
8      range", _FUNCTION_,
9      lp->ppp_slot);
10   return;
11 }
12 is = ippp_table[lp->ppp_slot];
13 isdn_ppp_frame_log('ccp-rcv',
14   skb->data, skb->len, 32,
```

Defect Report B:
File:
`/drivers/isdn/i4l/isdn_net.c`
Suspect Variable:
`isdn_dc2minor`

```
1  if (!lp->master)
2    qdisc_reset(lp->netdev->
3      dev.qdisc);
4  lp->dialstate = 0;
5  dev->st_netdev[isdn_dc2minor(
6   lp->isdn_device,
7   lp->isdn_channel)] = NULL;}
8  isdn_free_channel(
9      lp->isdn_device,
10     lp->isdn_channel,
11     ISDN_USAGE_NET);
12 lp->flags &=
13   ~ISDN_NET_CONNECTED;
```

Defect Report C:
File:
`/drivers/isdn/i4l/isdn_net.c`
Suspect Variable:
`isdn_dc2minor`

```
1   sidx = isdn_dc2minor(di, 1);
2  #ifdef ISDN_DEBUG_NET_ICALL
3   printk(KERN_DEBUG "n_fi: ch=0\n");
4  #endif
5
6   if (USG_NONE(dev->usage[sidx])){
7   if (dev->usage[sidx] &
8         ISDN_USAGE_EXCLUSIVE) {
9     printk(KERN_DEBUG "n_fi: 2nd
10      channel is down and bound\n");
11    if ((lp->pre_device == di) &&
12        (lp->pre_channel == 1)) {
```

Figure 3.1: Example Linux defect reports produced by Coverity's Static Analysis. The information presented is a mix of syntactic (e.g., the implicated code) and structural information (e.g., the suspected defective execution path and programmatic source of the defect). Syntactically, there are both similarities and differences between all three reports. When considering structural information, it appears that reports B and C share commonalities while A differs from both.

3.2 Motivation

Static defect reports typically contain both implicated source code lines and structural (or semantic) information related to the reported defect. The use of the term "semantic" to describe statically obtained information about potential defects here may be somewhat misleading, thus henceforth we describe such information as *structural* to avoid confusion. State-of-the-art duplicate detection techniques target human-written reports or source code respectively and do not perform well on, or are not applicable to, the information produced by static-analysis defect detectors (see Section 2.2.2

for more detail). Instead, our proposed technique exploits the structure of the information present in automatically generated defect reports to identify highly related clusters. In this section, we motivate this approach by presenting examples of automatically identified defect reports that are conceptually similar but exhibit syntactic discrepancies, and show how additional structural information can help us identify which of the example reports are related.

Coverity's Static Analysis tool ("Coverity SA") is a multi-language commercial bug finder that uses semantic path information to pinpoint likely bugs, matching known faulty semantic patterns [22]. Coverity SA reports bugs by outputting the type of error suspected and the given error's location in the code. We ran Coverity SA on version 2.6.15 of the Linux kernel and it reported over 1,500 candidate defects. We show three example reports from two separate files in Figure 3.1. We show only the file, suspected programmatic element, and implicated source lines (highlighted) in the context of surrounding code for ease of presentation.

There are several syntactic similarities to note in the given code. For instance, the immediate context of the code implicated by all three reports contain multiple references to a variable named `lp`. Additionally, reports A and C share calls to the `printk` debugging function while reports B and C share calls to the `isdn_dc2minor` function. However, there are obvious syntactic differences between all three reports as well. For example, parameters to the shared function calls are different across all three reports. While some might be tempted to group reports A and C due to the abundance of syntactic similarities, others might group reports B and C because of the similarity in function calls in the lines directly implicated.

As the syntactic information in the three reports fails to strongly indicate the presence or absence of conceptual similarity, we leverage the structural information in each report to get a more definite measure of relatedness. For example, we note that reports B and C are not only in the same file, but implicate the same programmatic element, `isdn_dc2minor`, as the source of the error. By comparison, report A shares few structural similarities with either reports B or C.

In reality, the `isdn_dc2minor` function called in reports B and C fails to check for negative values before returning and uses `-1` as its default return value. There are no corresponding checks for negative values at either call site shown before attempting to use the result to index arrays, and thus reports B and C represent actual defects. By contrast, the code in report A cannot suffer from the same type of error because an if statement ensures `lp->ppp_slot` is in bounds. A developer or tool using only syntactic information may conclude that the three defect reports are all related because of shared function calls and variables, or completely unrelated because of an abundance of unique program identifiers and code structure. However, the structural similarities between defect reports B and C provide more conclusive evidence that B and C describe a related defect while report A describes an orthogonal problem. We aim to leverage such information in our technique to identify such relationships quickly and automatically cluster defect reports that may be conceptually related but syntactically distinct.

Inspecting reports B and C aggregately can save time. For example, a check for negative values in B could be easily

adapted to work in C, or a patch might restructure negative return handling in `isdn_dc2minor` and thus address both call sites. Both candidate fixes represent potential time savings (i.e., either because the same negative value check can be inserted in two places or because one fix to the callee affects multiple callers) and thus clustering reports like these can reduce maintenance effort.

Such a clustering technique could be used during both report triage and defect fixing to expose similarities that may not be obvious from manual inspection. Using such a tool allows developers to triage and fix similar defects in parallel, thus saving maintenance effort overall.

3.3 Methodology

We propose a similarity model for automatically reported defects, allowing for the use of off-the-shelf clustering algorithms. Our model considers both syntactic and structural judgments of relatedness, using information reported by static analysis tools. We first outline the structure of an automatically produced defect report and describe how to extract the relevant pieces of syntactic and structural information (Section 3.3.1). Once we have obtained the structured information from the defect reports, we measure defect similarity by systematically comparing the sub-parts of defect reports by both adapting existing techniques and introducing novel similarity metrics (Section 3.3.2). We then learn a descriptive model of defect report similarity using linear regression (Section 3.3.3) and explain how to use the resulting similarity measures to then cluster related defect reports (Section 3.3.4).

3.3.1 Modeling Static Analysis Defect Reports

In Section 3.2 we introduced the Coverity SA bug finder; in practice there are many tools that report candidate software defects (e.g., [12, 97, 104, 127]). In this chapter we also focus on FindBugs [98], an open source, lightweight static bug finder that uses pattern recognition to find known faulty code sequences. Similar to Coverity SA, it reports the type and location of the suspected defect. A Coverity SA or FindBugs defect report can be viewed as 5-tuple $\langle \mathcal{D}, \mathcal{L}, \mathcal{P}, \mathcal{F}, \mathcal{M} \rangle$ where: $\mathcal{D}$ is a free-form string naming the *defect type*; $\mathcal{L}$ is a $\langle source\_file, line\_number \rangle$ pair representing the *line* directly implicated by the tool as containing the defect in question; $\mathcal{P}$ is a sequence of zero or more $\langle source\_file, line\_number \rangle$ pairs, encoding a static execution *path* of lines that may be visited when exercising the defect; $\mathcal{F}$ is a string naming the nearest enclosing *function*, class or file to $\mathcal{L}$; and $\mathcal{M}$ is a set of zero or more free-form strings holding any additional *meta-information* reported by the analysis (e.g., optional defect sub-types, categorical information for given lines of code, or suspected sources of the defect). Our technique operates on reports produced by any analysis that follow this format (or a subset of it, e.g., [104, 164]), regardless of defect-finding strategy.

The defect report components provide several potential sources of both structural and syntactic information that may be used in measuring the similarity between two reports. We use certain pieces of information exactly as they appear in

a defect report, and coerce others to maximize the utility of the information extracted. The following paragraphs detail the specific types of information used to measure report similarity.

Function — Taken verbatim from $\mathcal{F}$, this string represents the name of the nearest enclosing function of the line indicated to be the manifestation of the defect. When a defect is reported outside a function, we use the enclosing class or file.

Path Lines — This information is a sequence of strings representing the source code lines implicated in a static path that may be executed to reach the site of the defect ($\mathcal{P}$ in our model). We hypothesize that errors on the same or similar execution paths may be related. Beyond comparing these path sequences explicitly, we additionally sort the source lines in $\mathcal{P}$ alphabetically to help expose defects that implicate similar lines of code but in different orders.

Code Context — Given the exact line indicated in $\mathcal{L}$ as the manifestation of the bug, the code context is the sequence of strings representing the three preceding and three following lines as they appear in the original source file. This window of code is an approximation of the context of the bug. We hypothesize that defects that occur in similar contexts (e.g., inside a `try/catch` block) may be similar.

Macros — By extracting all tokens containing *only* capital letters or digits from the actual source line text referenced in both $\mathcal{L}$ and $\mathcal{P}$ above, this information approximates the set of macros referenced in any indicated code line. Finding the exact set of macros in code requires a preprocessor and is prohibitively expensive. We thus use an approximation: While some tokens that appear in all capital letters may not actually be macros, a random check of 20 such instances showed that 85% indeed were. We hypothesize that the use of the same macros may indicate similarity between defect reports.

File System Path — This information is a string representing the exact path of the indicated file (taken from $\mathcal{L}$) in the given project's file structure which attempts to link defects that are in the same module or even sub-folder. Similar to the enclosing function, we hypothesize that defect reports indicating locally-close files may exhibit similarity.

Meta-tags — When available, this is a set of strings taken directly from $\mathcal{M}$: any additional information from the static analysis tool. With respect to the defect reports presented in Section 3.2, this information includes the suspected source of the defect. Depending on the tool being used to find defects, the type and amount of information can vary widely. We hypothesize that *any* information produced by the static analysis tools *may* be useful when measuring similarity.

3.3.2 Defect Report Similarity Metrics

We propose a set of lightweight similarity metrics for tool-reported defects $\langle \mathcal{D}, \mathcal{L}, \mathcal{P}, \mathcal{F}, \mathcal{M} \rangle$ that are collectively applicable for both syntactic and structural information. Since we are interested in relationships *between* defect reports, the basic unit over which we measure similarity is a *pair* of defect reports. We determine an overall similarity rating for

two defect reports by computing a weighted sum (Section 3.3.3) of the similarities of their individual sub-components (described in Section 3.3.1). This similarity model allows us to *cluster* related defect reports (Section 3.3.4).

We use metrics from information retrieval and natural language processing in addition to introducing novel lightweight similarity metrics specifically applicable to the structure of the information present in this domain to compare individual defect report sub-components. Unless otherwise specified, we tokenize raw strings by splitting on whitespace and punctuation. The metrics we consider are described in the following paragraphs.

Exact Equality — a character-wise boolean match of two strings. Intuitively, reports with exactly-matching sub-components are likely related.

Strict Pairwise Comparison — the percentage of tokens from two strings that match exactly (comparing a_i to b_i for two token sequences a and b). When comparing textual lines of code, for instance, this metric can identify similar code that differs only in a few variable names or method calls.

Levenshtein Edit Distance — adapted from the information retrieval community, this metric in an approximate string matching technique measuring the number of incremental changes necessary to transform one string into another [165]. We lift the traditional metric, which operates on strings of characters, to sequences of tokens. Working over the alphabet of all tokens in either string, we count the number of token-level changes to transform one string into the other. Levenshtein distance relaxes a strict pairwise comparison, allowing approximate alignments. Spell checkers often use a similar method for suggesting replacement words for misspellings. Conceptually, our lifted Levenshtein distance is similar: it suggests defect reports with information that may be related.

TF-IDF — a document similarity metric common in the natural language processing community. It rewards tokens unique to the two documents in question and discounts tokens that appear frequently in a global context [166]. The use of TF-IDF assumes the existence of a representative corpus from which to measure the relative global frequency of all tokens. We take as this corpus the set of all tokens from all tool-produced defect reports to be clustered for a given program. We compare two documents by inspecting the *term frequency* (tf) and *inverse document frequency* (idf) of each token individually. *Term frequency* measures the relative count of all words while *inverse document frequency* measures the "uniqueness" of terms and discounts common words like *int* or *the*, while weighting unique and thus potentially more meaningful words highly. For example, referring back to the defect reports presented in Section 3.2, the token `isdn_dc2minor` occurs in both reports B and C, but only in 0.69% of reports overall. Sharing this rare token increases the TF-IDF measure between these two reports, exposing an inherent similarity. By contrast, the token `lp` occurs frequently in all three reports and in 4.95% of others for Linux: it thus has a lower idf value. This prevents TF-IDF from mistakenly indicating as similar all defect reports with this globally-frequent token.

Largest Common Pairwise Prefix — the number of tokens two strings have in common when comparing each from left to right (i.e., the largest i such that $\forall j.\ 0 \leq j \leq i \implies a_j = b_j$). To illustrate the utility of this metric, consider two statements that assign the results of similar function calls to the same variable. Even if the function calls'

parameters differ, this metric will capture the initial similarity between the two lines. Put differently, the way programs are written sometimes corresponds loosely to English, where the subject and verb usually appear towards the beginning of a sentence. Similarly, the left-most columns in many high-level programming languages (e.g., generally, the variable being assigned to or the root object of a method call) are the most fundamental to the execution and state of a program. For these reasons, we hypothesize that checking for similarities between the prefixes of code-based information might expose related defect reports.

Punctuation Edit Distance — a lightweight metric for structural code similarity. Traditional methods, like comparing control flow graphs, are expensive and are made difficult because compilation may be not be available on all projects during the triage stage. We instead adopt a lightweight metric that approximates program structure while retaining consideration for the sequence in which programmatic events occur. We compute the Levenshtein edit distance between token sequences with all non-punctuation removed (e.g., only curly braces, parentheses, operators, etc. remain). As an example of the utility of such a metric, consider two pieces of code that share both the same method calls and similarly structured loops. A similar pattern of parenthesis, commas, curly braces, and semicolons will help make the relatedness evident. By abstracting away textual identifiers, this metric complements more language-focused notions.

These metrics operate on a variety of input types. We coerce one type of information to another when necessary. For example, any string or set of strings can be viewed as a "bag of words" (the document data structure used by TF-IDF) by splitting on punctuation and whitespace while aggregating term frequency counts. Similarly, a set of strings can be coerced into a sequence (used by Levenshtein edit distance, for instance) by sorting them in order of textual appearance or alpha-numerically.

3.3.3 Modeling Report Similarity

The textual code-based and structural programmatic features outlined in Section 3.3.1 serve as input to the similarity measurements, allowing us to compare sub-components of two automatically generated defect reports. We apply each similarity metric to all pairs of applicable report sub-components to obtain similarities for each pair of reports. We elide combinations with little or no predictive power for simplicity.

We avoid asserting an *a priori* relationship between these measurements and whether a pair of defect reports are related. Instead, we build a classifier that examines a candidate report pair and, based on a learned linear combination of weighted feature values, determines whether the pair is "similar." Thus, the similarity judgment for a pair of defect reports is a sum of weighted features (where each f_i is similarity metric value for a pair of report sub-components):

$$c_0 + c_1 f_1 + c_2 f_2 + \ldots + c_n f_n > c' \tag{3.1}$$

Two defects are called "similar" if the resulting aggregate sum is greater than an experimentally chosen cutoff: c'. We use linear regression to learn values for c_0, c_1 through c_n, and c'. A training stage, detailed in Section 3.4, is required to learn this classifier. We choose a linear model to allow for exploration of a series of smooth cutoffs given a single model.

3.3.4 Clustering Process

A cluster of defect reports wherein each individual report is "similar" (with respect to our model) to all others is amenable to aggregate triage. Having defined similarity between defects reports, we now require a lightweight and accurate method for clustering related defects. Traditional clustering techniques (e.g. k-medoid clustering) often try to measure the similarity between single entities given a formal metric space. Specifically, k-medoid clustering assumes that all features are real-valued and weighted equally in the model. First, we do not assume that the features in our model warrant equal weighting (as evidenced by the learned coefficients in our similarity model). Additionally, our features are not real-valued measures of an individual defect's properties, but rather relative measures of similarity. We thus adopt a well-known algorithm for measuring interconnectedness of components for the purpose of clustering.

One can view a cluster of similar defects as an undirected graph where the vertices represent defects and the edges represent the similarity relationship (that is, any connected vertices are considered "similar" using equation (3.1)). To prefer accurate clusters and avoid falsely clustering unrelated defects, clustering can be performed by finding maximum cliques in the induced graph [167]. Finding cliques ensures that any defect in the clique (cluster) will be similar to all other defects therein.

We propose a two-phase recursive approach to clustering:

1. Construct an undirected graph where the vertices represent all remaining, unclustered defects and the edges signify our definition of "similarity."

2. Find the maximum clique, output all included defects as a cluster and remove them from the graph; return if no defects remain, otherwise recurse.

In the worst case, clique finding requires exponential time: the time complexity is $\mathcal{O}(n \times 2^n)$ where *n* is the number of vertices in the overall graph. In practice, "almost-cliques" are rare (i.e., spurious interconnecting edges between clusters are sparse when a high "similarity" cutoff is chosen) and our implementation runs sufficiently fast. For example, on a Linux kernel module with 869 defect reports, the average run time was 0.088 seconds. This approach to clustering produces distinct sets of defects that display a high degree of internal similarity, as we show in the next section.

Program	Version	KLOC	Reports	Description
Blender	2.45	996	827	3D content creation suite
GDB	6.7	1,689	827	Multi-language debugger
Linux (fs)	2.6.15	521	175	Linux OS Filesystem module
Linux (sound)	2.6.15	420	869	Linux OS Sound module
Linux (other)	2.6.15	4,263	214	All other Linux OS modules
MPlayer	1.0rc2	845	500	Media player
Perl	5.8.8	430	63	Perl language interpreter
Ruby	1.8.6-p111	194	75	Ruby language interpreter
Xine	1.1.10.1	499	292	Media player
	Totals:	**9,862**	**3842**	
Bcel	5.1	56	238	Byte Code Engineering Lib
Eclipse	3.1.2	3,618	4345	Programming IDE
JFreeChart	1.0.1	211	338	Chart toolkit
Spring	2.0.8	430	185	Java application framework
	Totals:	**4,316**	**5106**	

Table 3.1: Test programs and defect reports used to evaluate our algorithm. The top group of programs are written in C while the bottom group is written in Java. Note: the KLOC totals represent the number of lines analyzed by the bug finders and might be smaller than the total number of lines in the projects.

3.4 Evaluation

We seek to evaluate our technique's utility when clustering defect reports and also put it in context with relevant work. We thus address four research questions:

- *R1*: How effective is our technique at accurately clustering defect reports produced by off-the-shelf static analysis tools?
- *R2*: Does our approach outperform existing code clone detection techniques when clustering defect reports?
- *R3*: How does our technique perform across different static analysis tools and different languages?
- *R4*: Do humans agree with the clusters produced by our technique?

For the purposes of these experiments, we collected defect reports from eleven C and Java programs comprising over 14 million lines of code and yielding over 8,000 defect reports from Coverity SA and FindBugs. Further details of these benchmark programs can be found in Table 3.1.

3.4.1 Learning a Model

First, we construct a model that, given a set of similarity measurements between the sub-components of two candidate defect reports, determines whether the two reports should be considered highly related. We use linear regression to learn the coefficients c_i and the cutoff c', such that the model declares two reports similar according to equation (3.1).

Linear regression requires training data consisting of the measured features annotated with the response variable (i.e., the "correct" answers). The response variable for our model is defect report similarity — a human judgment. Because such a judgment cannot be automatically measured, we hand-annotated all combinations of defect report pairs (where only defect reports of the same "type" could be potentially clustered) to serve as a ground truth when training and testing the model. Our goal is to cluster not just syntactically similar defect reports, but also those that are related semantically. When annotating the data set, we therefore deemed two defect reports "similar" if any of the following criteria were met:

1. the code contexts displayed significant syntactic similarity while implicating the same defect
2. the implicated code for both reports was semantically related such that the underlying causes of the defects were the same
3. the reported defects' code exhibited semantic similarities such that the defects would manifest in the same way

We mitigate the threat of over-fitting our model by specifically training and testing on different sets of data. We randomly selected small subsets of the annotated defect report pairs for each benchmark program for the purpose of training. We then tested the model on the remaining data. As such, we gain confidence that our model is not simply encoding specific data points, but rather learning meaningful weights for the associated sub-component comparisons as intended.

An advantage of our approach is that it does not distinguish between "true positive defect reports" (real bugs) and "false positive defect reports" (spurious reports from the static analysis tool). When clustering reports to expedite triage, effort can be saved in both cases: false positives must be identified as such, and doing so aggregately saves maintenance effort.

3.4.2 Maintenance Savings versus Cluster Accuracy

The goal of our technique is to reduce maintenance effort by clustering tool-generated defect reports, allowing developers to triage and even fix defects in aggregate. In this section, we evaluate the potential for effort savings associated with our tool (*question R1*). Additionally, we put our tool in context by comparing it with the closest related duplicate detection techniques (*question R2*).

Metrics

To evaluate both research questions, we use two distinct success metrics. First, we measure the average internal accuracy of all clusters produced. That is, for each proposed cluster we measure the ratio of the size of the largest contained clique (with respect to our ground-truth annotations) to the size of the cluster as produced by our technique. For

example, a cluster of size five where only four reports are perfectly interrelated would have an accuracy of 0.8. Second, we compute the percent reduction in size of the overall set of defect reports when using the resulting clustering to handle and triage clustered defect reports aggregately. We "collapse" each emitted cluster into one effective defect report, assuming (given the stated definition of *related* defect reports) that similar reports can be handled in parallel. For example, if there are 20 original reports and an approach identifies two clusters of size five each, the resulting *effective* size is 12 conceptual clusters (10 singletons and 2 of size 5), making 40% reduction in the number of reports that must be considered separately and addressed using separate reasoning. We recognize that not all defects take the same amount of time or human effort to triage and fix and thus note that approximating the reduction in human effort based on the reduction in defect reports would be strictly an estimation. Gauging the effort needed to triage and fix any one defect [168] is orthogonal to this work and, as such, we simply measure a reduction in the number of reports for our evaluation.

Code clone tools

To our knowledge, there are no existing fully-automatic techniques for clustering defect reports produced by static bug finding tools. Kremenek *et al.* propose a defect report ranking technique based on clustering, but it relies on repeated human feedback and thus is not directly comparable to our technique [38]. However, code clone detection is a closely related task — reports implicating similar code (e.g., from copy-and-paste development, or just from similar development logic) may likely be related, and thus we can use such techniques as a baseline for comparison. Tool-generated defect reports contain an abundance of code-based information and thus, adapting code clone detection tools for this task provides a direct means of comparison. Additionally, code clone tools rely almost exclusively on syntactic string matching techniques and as such provide an excellent baseline for comparison: any increase in accuracy or the number of clustered defect reports exhibited by our tool can be attributed to our inclusion of structural information or use of diverse similarity metrics.

There are many state-of-the-art techniques capable of performing clone detection with high accuracy; we adapt three popular tools to compare against our technique: ConQAT, PMD, and Checkstyle [161, 162, 163]. These tools typically take as input a set of source files and produce a list of all code clones. We adapt them to defect report clustering by creating a set of synthesized source files, each deriving from an individual report. For a given defect report, we construct a synthesized source file by concatenating $\mathcal{L}$ (the source line implicated) and $\mathcal{P}$ (the implicated execution path source). The set of all synthesized files (corresponding to all defect reports in question) is used as input for the given clone detection tool, and we then use the code clone tools' output as a defect report similarity metric and perform clustering in the same manner as our technique (as described in Section 3.3.4).

Results

Figure 3.2 and Figure 3.3 show the percent reduction of the overall set of defect reports when using each clustering approach at varying levels of cluster accuracy, split between C and Java defect reports. These results are presented in terms of Pareto-optimal frontiers to show the tradeoff between cluster accuracy and the number of distinct defect reports. Each point on a Pareto frontier represents a possible outcome for a parametric technique. Our approach admits more fine-grained adjustment than off-the-shelf code clone tools as it is parametric based on modeled similarity and not simply the size of the matches found in the code.

Our technique clusters more defects than comparable code clone detection techniques at nearly all levels of accuracy for both languages. When considering Java defect reports, our technique outperforms all code clone tools at all levels of accuracy. Additionally, our technique is capable of perfect accuracy (the bottom right portion of either graph), while the other tools are not. We note that while lower accuracies appear to yield large clusters and thus a great reduction of the overall set of defects, in practice, spurious reports in such clusters would greatly reduce the benefit of treating such defects in aggregate. We assert that higher levels of accuracy should be favored to reduce maintenance effort. We present the full spectrum of accuracy values for the sake of completeness.

Comparing the area under competing Pareto frontier curves provides a way to generalize performance across all tradeoffs. When considering all defect reports, the area under the curve for our technique is 1.4 and 2.5 times larger than ConQAT and PMD, the two multi-language code clone tools we consider, respectively (Checkstyle works only for Java programs and thus is not considered here). Code clone tools take into account mostly syntactic features, thus the increase in performance associated with our tool can likely be attributed to the inclusion of structural, semantically-related, features. We believe that the disparity in performance between the code clone tools and our techniques can be explained by clusters of conceptually similar but syntactically unique defect reports (see Section 3.2).

With the goal of usability in mind, we desire high-quality results that humans may actually be able to use to triage and potentially fix bugs in practice. At a level of 95% accuracy we find that we can cluster 60.6% of similar defects in practice (based on the hand-annotated data set presented in Section 3.4.1. This result is a weighted average of defects from the benchmark programs in both C and Java. This achieves the usability goal set forth in Section 1.4.

3.4.3 Semantic Clustering Generality

Having demonstrated that our technique can reduce the number of distinct defect reports (e.g., to save developer maintenance effort), we investigate the differences in performance across languages and when considering defect reports produced by different static bug finding tools (*question R3*). We show that our tool is general in nature, which is one of the overarching goals of this dissertation.

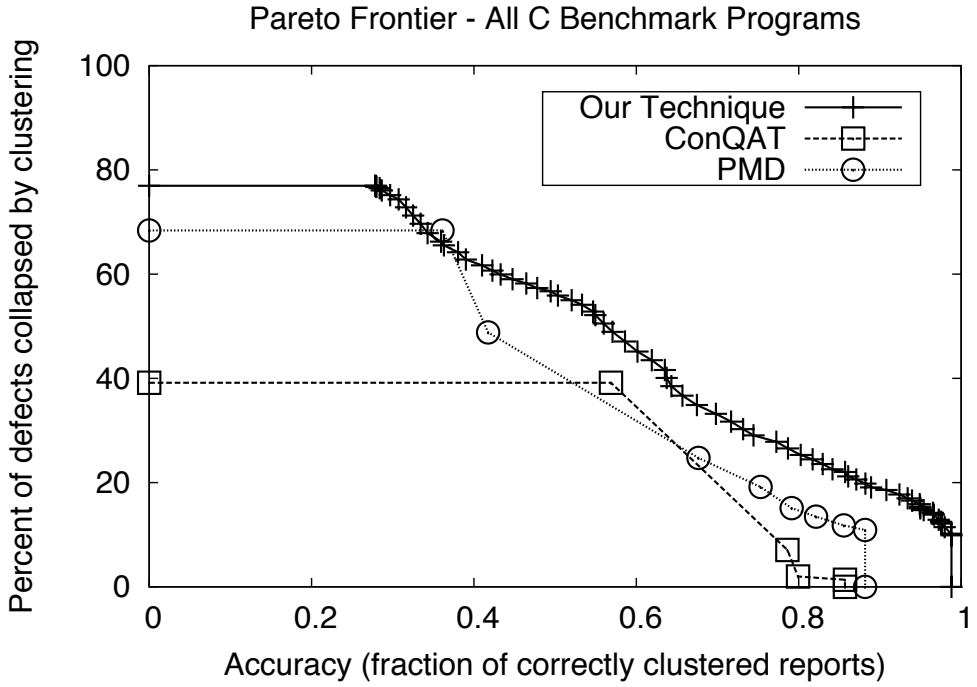

Figure 3.2: Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate reduction in the number of defect reports from clustering for C benchmark programs.

Different languages

While syntax-based code clone detectors exhibit varying levels of performance across languages, our technique generalizes with higher stability. Notably, the Pareto frontiers for our tool with respect to both C and Java defect reports share a similar shape and comparable levels of defect report set reduction at varying levels of accuracy while those of the code clone tools do not. Our model does not use language-specific features, and thus displays cross-language consistency. On average, ConQAT and PMD (the two code clone detectors that work on both C and Java code) show over 5 times more variance (in terms of under-curve area) across languages as compared to our technique.

Different tools

There are numerous qualitative differences between the defect reports produced by Coverity SA and FindBugs, thus it is not obvious that a given clustering technique will immediately generalize across static analyzers. Coverity SA yields semantically-rich data, typically producing non-empty $\mathcal{P}$, non-empty $\mathcal{F}$, and various additional information in $\mathcal{M}$. FindBugs, by contrast, often produces fewer suspected lines and even generalizes some types of defects to only

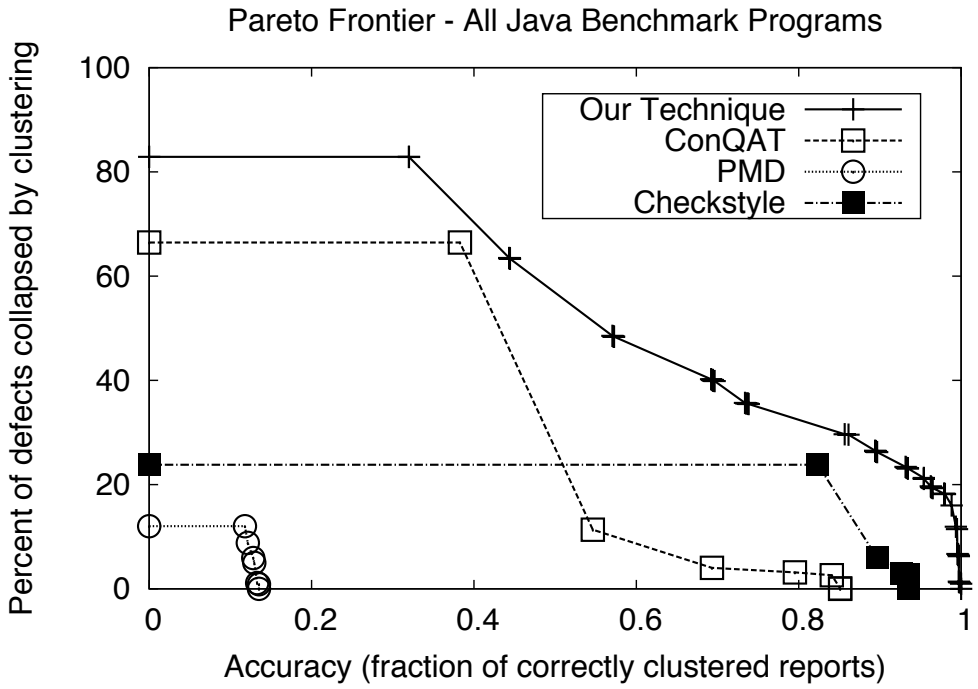

Figure 3.3: Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate reduction in the number of defect reports from clustering for Java benchmark programs.

the containing class, yielding $\mathcal{L}$ as the only line of code defining the associated defect. Similar to the cross-language comparison, our technique performs comparably on Java defect reports from both Coverity SA and FindBugs (FindBugs is not meant to run on C code). ConQAT, PMD, and Checkstyle exhibit variance comparable to that of our technique across different static bug finding tools, but our technique emits larger clusters (thus allowing for greater maintenance effort savings) at all levels of accuracy for both tools' defect reports. This further suggests that our semantically-rich tool is better suited to clustering multiple types of automatically produced defect reports than are the most closely related code clone techniques.

Predictive Power of Structural Metrics

We hypothesize that the inclusion of structural features accounts for the relatively high performance of our technique. Table 3.2 presents the features used in our model along with the corresponding relative predictive power (or "quality measure") of each as measured by the ReliefF method [169, 170]. ReliefF does not assume linear independence of features and thus is appropriate given that some of our model's features may overlap as they derive from similar parts of

Information	Match Type	ReliefF
Path Lines	Strict Pairwise	0.0043
Code Context	Strict Pairwise	0.0042
File System Path	Common Prefix	0.0039
Code Context	Levenshtein	0.0022
Path Lines	TF-IDF	0.0021
Path Lines	Common Prefix	0.0020
Path Lines	Punctuation	0.0016
Path Lines, Sorted	Common Prefix	0.0016
Macros	TF-IDF	0.0008
Path Lines, Sorted	Levenshtein	0.0009
Path Lines	Levenshtein	0.0009
Meta-tags, Sorted	Levenshtein	0.0003

Table 3.2: A list of the predictive power of the similarity features used by our technique. The "information" column notes the part of the static analysis output being examined and the "match type" column indicates the type of similarity metric used. Features' qualities are measured relative to one another, where higher values indicate more predictive power.

the code or defect reports. ReliefF reports each feature's importance based on the *relative* magnitude of each feature's quality measure — larger numbers indicate more powerful features. Notably, some of the most predictive features use parts of the defect reports including the code path, the contextual window around the suspected defect, and the file system path, none of which are used by the code clone tools. This suggests that the use of structural information is beneficial when clustering duplicate automatically generated defect reports. Macros and meta-tags proved to be weaker sources of information: we hypothesize that because not all defect reports contain this information, they may not be universally powerful predictors.

3.4.4 Cluster Quality

Clustering defect reports is advantageous only if the clusters contain reports that are, in fact, related. An incorrectly clustered set of defect reports that is mistakenly triaged in the same way may negate some or all of the maintenance effort savings associated with clustering. We must verify that our clustering technique agrees with human maintenance judgments (*question R4*).

Our technique models a human cognitive notion of defect report similarity and thus any qualitative validation cluster accuracy should include human judgment. In Section 3.4.2 we quantitatively show that our technique is capable of achieving high accuracy with respect to our human-annotated data set. In this subsection, we present evidence from a developer survey showing that our annotated data set is grounded in reality and thus that developers may benefit from using our approach.

Our survey goal was to evaluate our annotation technique and provide confidence that it generalizes. Focusing on clusters that should and should not be triaged together in practice, we presented 12 developers (graduate students and

developers from industrial firms) from both academia and industry with 50 clusters of defect reports. We randomly selected 25 "accurate" clusters from those produced by our technique at accuracy greater than 90% (manually verified against the hand-annotated data set). We also randomly selected 25 "inaccurate" clusters selected from those produced by PMD with accuracy less than 10%. For a given cluster, participants were provided with the type of defect being clustered and the code implicated in all related reports. Participants were shown all 50 clusters in a random order and asked to determine whether they believed the reports in a given cluster could be triaged and potentially fixed in the same way — that is, were the clustered reports likely representative of the same or highly related bugs? This high-level definition for the "similarity" of defect reports mimics the stated use case for our technique.

We hypothesize that humans would strongly agree with our annotations, thus validating the results presented in Section 3.4.2. We present results in terms of both raw agreement percentages and Randolph's free-marginal multirater kappa, an aggregate measure of inter-annotator agreement that does not assume a fixed distribution of categorizations for a given participant [171]. Free-marginal multirater kappa values range from -1.0 to 1.0, where a value of 1.0 represents perfect agreement and values greater than 0.8 indicate "*strong*" agreement. Participants agreed with our annotations with respect to "accurate" clusters 99% of the time (with a free-marginal multirater kappa agreement of 0.96), suggesting that our annotation process is grounded with respect to human judgments of defect report similarity. Conversely, participants showed more variability with respect to the PMD-generated clusters containing defect reports we annotated as "not accurate." Participants only agreed that clusters we annotated as "inaccurate" (or not related) were, in fact, related 44% of the time (free-marginal multirater kappa agreement of 0.28). This finding argues for a parametric technique such as ours: when high accuracy is demanded of our tool, humans show almost perfect agreement with it, but since human variability exists, some developers may prefer looser, and thus potentially larger, clusters.

We have shown that developers may prefer different levels of accuracy for defect report clustering and that our technique is capable of near-perfect accuracy. However, the increase in clustering size grows rapidly as small accuracy decreases are allowed. For instance, while the number of defects effectively removed from the overall set at 100% accuracy is 4.30%, at 95% accuracy, the savings jumps to 18.35% (an $4\times$ increase). By contrast, all three code clone tools fail to ever achieve 95% accuracy. The example cluster presented in Section 3.2 (i.e., a cluster containing defect reports B and C) was produced by our technique tuned to an accuracy level of 85%, further suggesting that perfect accuracy is not required for useful clustering.

3.4.5 Cluster Case Study

To further explore the quality of the clusters produced by our technique (*R4*), we present an example cluster of defect reports from the Eclipse project in Figure 3.4. The three defect reports presented are categorized as "forward null"

defects, which suggests that the successful execution of a given statement necessarily indicates that the value of a specific variable in a following statement will be `null`.

Through careful inspection, we have concluded that these defects are not only false positives, but are also similar enough to be handled aggregately, saving maintenance effort. Notably, the Eclipse-specific `Assert.isTrue(...)` method called in three cases will throw an `AssertionFailedException` if the variable in question, `entry`, is ever null. This will interrupt execution, preventing the suspected defective lines from executing. Coverity's Static Analyzer is equipped with functionality to handle such system-specific idiosyncrasies, but has to be manually configured to do so. These three defect reports are additionally similar because the false positive is caused by a call to `Assert.isTrue(...)` and concerns a variable with the same name, created from the same source method call in all three cases. In practice, a developer presented with this cluster could quickly identify the commonalities and discard these three defect reports aggregately, reducing the required maintenance effort.

Additionally, these reports exemplify the utility of structural features and some of the shortcomings associated with syntactic-only models. All three defect reports exhibit the following structural similarities:

1. Spatial Locality — All three machine generated defect reports indicate code in the CheatSheetStopWatch.java file within the UI module of the Eclipse code base.

2. Contextual Similarity — Each suspected defective code path is immediately preceded by two textually-identical lines of code (lines 2–3 in all three examples).

3. Punctuation Edit Distance — The indicated lines exhibit high similarity with respect to punctuation. For instance, lines 4 from each of the reports are identical when only punctuation is considered.

However, there are syntactic differences between these three defect reports that may make it difficult for syntax-only approaches to definitively indicate similarity:

1. While the statements spanning lines 4 through 6 in each report exhibit some similarities, the large string literals are unique in each case.

2. In all three cases, the suspected location of the respective errors (the last highlighted line in each code segment) are very different. Specifically, report X indicates an assignment statement, report Y an assertion, and report Z a conditional.

Our technique produced this cluster with perfect clustering accuracy and a 3.25% reduction of defect reports overall. Thus, a user requiring even the highest level of accuracy would be provided this cluster in practice. By contrast, only one of the code clone detection tools, ConQAT, also produced this cluster — at a level where its accuracy was 0.30. At this level of accuracy, 70% of reports clustered using ConQAT would be miscategorized, and much of the effort savings associated with clustering reports would be lost.

Defect Report X:

```
1  public void stop(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start !=-1,
5     "start() must be called before usingstop()");
6    entry.stop = System.currentTimeMillis();
7  }
```

Defect Report Y:

```
1  public long totalElapsedTime(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start !=-1,
5    "start() must be called before using
6    totalElapsedTime()");
7    Assert.isTrue(entry.stop != -1, "stop() mustbe
8    called before usingtotalElapsedTime()");
9      //$NON-NLS-1$
10   return entry.stop - entry.start;
11 }
```

Defect Report Z:

```
1  public void lapTime(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start !=-1,
5    "start() must be called before usinglapTime()");
6    if(entry.currentLap == -1) {
7      entry.previousLap = entry.start;
8    } else {
9      entry.previousLap = entry.currentLap;
10   }
11   entry.currentLap = System.currentTimeMillis();
12 }
```

Figure 3.4: Three defect reports from Coverity Static Analysis when run on version 3.1.2 of the Eclipse IDE. The highlighted lines are specifically implicated by Coverity SA as the suspected defective execution path while the additional lines provide context. In each case, the last highlighted line is the exact spot of the suspected defect.

3.5 Threats to validity

While our experiments were designed to show the utility of our technique when clustering defect reports produced by different static analysis tools over large, open-source programs in several languages, our results may not generalize to industrial practice. First, our benchmark programs may not generalize to all industrial code. To mitigate this threat, we selected both large and small programs from varying domains spanning both C and Java. In addition, many of these benchmarks are used by Coverity for in-house testing, suggesting external belief in their generality.

Additionally, Coverity SA and FindBugs may not generalize to all static bug finders and thus our technique's performance may not generalize to all such tools. We attempted to mitigate this threat by designing our technique to

operate on an abstract representation of defect reports and by testing our technique on two tools. Coverity SA is a commercial static bug finder that is semantically-rich and works across several languages. Comparatively, FindBugs is an open source pattern-based bug finder that is targeted at Java.

Finally, our method of manually annotating defect reports with respect to similarity may not generalize to the philosophy of all developers or systems. As noted in Section 3.4.1, defect report similarity is inherently a human judgment and thus different developers may have more or less strict ideas for what constitutes "similar" defect reports. We attempt to mitigate this threat in two ways. First, we designed our technique such that accuracy is adjustable parameter. Secondly, we asked multiple developers to assess on clusters produced by both our technique and a code clone tool. For clusters that we annotated as being "accurate" (thus, the defect reports are "similar"), developers agreed with our judgment 99% of the time.

3.6 Conclusion

This chapter presents a language-independent technique for clustering defect reports produced by static analysis-based bug finding tools. To the best of our knowledge, there are no existing tools specifically designed for such a task (e.g., tools for human-written reports focus instead on natural language), and we show that our tool is capable of clustering similar defect reports accurately to save maintenance effort. Our evaluation includes over 8,000 defect reports on over 14 million lines of code.

A quantitative evaluation shows that our tool outperforms state-of-the-art code clone tools adapted to the task of defect report clustering at nearly all levels of cluster accuracy. Additionally, our tool generalizes across defects found by both Coverity's Static Analysis tool and FindBugs in both C and Java programs. These results suggest that syntax-only approaches, like those used to find duplicate manual defect reports and code clones, are insufficient for the task of accurately clustering automatically generated defect reports. Developers could use such a clustering technique when attempting to triage and fix defects to save maintenance effort by handling similar defect reports aggregately.

Our technique can cluster defects of all types (with respect to the data set presented in Table 3.1), enhancing its generality. As part of a comprehensive evaluation, we also show that real world developers agree with our notion of an "accurate" cluster 99% of the time, thus suggesting our fully-automatic (i.e., requiring no additional human intervention) technique could be usable in practice. Furthermore, there is developer disagreement over "inaccurate" clusters, supporting our design decision that cluster accuracy be a tunable parameter. As bug-finding tools grow in popularity, processing their voluminous output becomes an increasing challenge: this chapter presents, to our knowledge, the first technique for clustering tool-generated defect reports and argues that it is effective.

The clustering technique presented in this chapter can help to facilitate the bug reporting and triage process. Chapter 4 describes a technique for automatically patching bugs, once they are found and triaged.

Chapter 4

Leveraging Program Equivalence for Adaptive Program Repair

"I have not failed. I've just found 10,000 ways that won't work."

– Thomas Edison

4.1 Introduction

THE previous chapter described a defect report clustering technique to reduce the cost of bug reporting and triage. Once a bug is found, reported, and triaged, a common next step is to fix the underlying problem to more closely align the software's implementation with its specification. This process has historically been carried out manually and can dominate the software lifecycle — a 2013 Cambridge University study finds that software developers spend 50% of their programming time "fixing bugs" or "making code work" [172]. To emphasize the cost of fixing bugs, we note that human developers take 28 days, on average, to address security-critical defects [14], and new general defects are reported faster than than developers can handle them [46]. This chapter introduces a new automatic bug fixing technique to help mitigate these costs.

Since 2009, when automated program repair was first demonstrated on real-world problems (ClearView [39], GenProg [129]), interest in the field has grown steadily, with multiple novel techniques proposed (AutoFix-E [130], AFix [26], Debroy and Wong [138], etc.) and an entire session at the 2013 International Conference on Software Engineering (SemFix [137], ARMOR [136], PAR [58], Coker and Hafiz [30]). We categorize program repair methods into two broad groups. Some methods use stochastic search or otherwise produce multiple candidate repairs and then validate them using test cases (e.g., GenProg, PAR, AutoFix-E, ClearView, Debroy and Wong, etc.). Others

use techniques such as synthesis (e.g., SemFix) or constraint solving to produce a single patch that is correct by construction (e.g., AFix, etc.). We use the term *generate-and-validate program repair* to refer to any technique (often based on search-based software engineering) that generates multiple candidate patches and validates them through testing. Although generate-and-validate repair techniques have scaled to significant problems (e.g., millions of lines of code [47] or Mozilla Firefox [39]), many have only been examined experimentally, with few or no explanations about how difficult a defect or program will be to repair.

A recent example is GenProg, which takes as input a program, a test suite that encodes required behavior, and evidence of a bug (e.g., an additional test case that is currently failing). GenProg uses genetic programming (GP) heuristics to search for repairs, evaluating them using test suites. A repair is a patch, edit or mutation that, when applied to the original program, allows it to pass all test cases; a *candidate repair* is under consideration but not yet fully tested. The dominant cost of such generate-and-validate algorithms is validating candidate patches by running test cases [173].

In this chapter, we provide a grounding of generate-and-validate automated repair, and use its insights to improve performance and consistency of the repair process. We first present exploratory research that investigates the possibility of further honing GenProg's GP parameters to speed up the repair process within the biology-inspired framework. This work shows that the most impactful bottlenecks seem to lie outside of the genetic-algorithm-based parts of the existing framework and thus we focus our efforts accordingly. We present a formal cost model, motivated in part by our categorization: broadly, the key costs relate to how many candidates are generated, and how expensive each one is to validate. This model suggests an improved algorithm for defining and searching the space of patches and the order in which tests are considered. Intuitively, our new algorithm avoids testing program variants that differ syntactically but are semantically equivalent. We define the set of candidate repairs as a quotient space (i.e., as equivalence classes) with respect to an approximate program equivalence relation, such as one based on syntactic or dataflow notions. Further optimizations are achieved by eliminating redundant or unnecessary testing. By recognizing that a single failed test rules out a candidate repair, our algorithm uses test suite prioritization to favor the tests most likely to fail (and the patch most likely to succeed) based on previous observations. The result is a deterministic, adaptive algorithm for automated program repair backed by a concrete cost model.

We also highlight a duality between generate-and-validate program repair and mutation testing [56], explicitly phrasing program repair as a search for a mutant that passes all tests. Examining the hypotheses associated with mutation testing sheds light on current issues and challenges in program repair, and it suggests which advances from the established field of mutation testing might be profitably applied to program repair.

Based on these insights, we describe a new algorithm and evaluate it empirically using a large dataset of real-world programs and bugs. We compare to GenProg as a baseline for program repair, finding that our approach reduces testing costs by an order of magnitude.

The main contributions of this chapter are as follows

- An assessment of the existing GenProg framework, paying particular attention to which genetic programming parameters might be altered to reduce the overall cost of producing patches. The results of this exploratory research lead to discoveries motivating a non-genetic-based approach.
- A detailed cost model for generate-and-validate program repair. The model accounts for the size of the fault space, size the *fix space* (Section 4.4), the order in which edits are considered (repair strategy), and the testing strategy.
- A technique for reducing the size of the fix space by computing the quotient space with respect to an approximate program equivalence relation. This approach can use syntactic and dataflow analysis approaches to reduce the search space and has not previously been applied to program repair.
- A novel, adaptive, and parallelizable algorithm for automated program repair. Unlike earlier stochastic repair methods, our algorithm is deterministic, updates its decision algorithm dynamically, and is easier to reason about.
- An empirical evaluation of the repair algorithm on 105 defects in programs totaling over five million lines of code and guarded by over ten-thousand test cases. We compare directly to GenProg, finding order-of-magnitude improvements in terms of test suite evaluations and over five times better dollar cost.
- A discussion of the duality between generate-and-validate program repair and mutation testing, formalizing the similarities and differences between these two problems. This provides a lens through which mutation testing advances can be viewed for use in program repair.

4.2 Exploring bottlenecks in the GenProg framework

This section describes research aimed at improving GenProg's expressive power. We present the result of experiments relating test cases to fitness functions in the context of GenProg. While this initially was designed to improve fitness functions, but ultimately led us to reconsider the framework entirely.) We discuss this work both to motivate and shape the new program repair framework.

4.2.1 Background and current state of the art GenProg framework

Previous results show that the existing GenProg framework is successful at fixing about 50% of the sampled bugs in past evaluations [47]. While these results represent a promising initial effort in the area of generic patch generation, further decreasing the cost of the algorithm would help to increase its efficacy in practice. Exploring GenProg's genetic parameters (e.g., crossover and mutation rate) has proven effective at generating additional defect repairs in the past [174]. While the focus of this work is cost reduction rather than expressive power, we hypothesize that further

improvement to the genetic algorithm's fundamental components may yield positive results in both arenas. Thus, we also investigated possible improvements to GenProg's fitness function to provide the algorithm with a more precise signal in the hope that such enhancements would speed up the search process.

The fitness function represents the objective function in the GenProg search framework, measuring the desirability (e.g., correct functionality) of a candidate program variant [50]. Useful individuals can then persist into further generations while unhelpful mutants can be discarded. With respect to GenProg, we desire program mutations that embody as much of the required functionality as possible. As such, GenProg has traditionally counted the number of test cases (both regression tests and those encoding buggy behavior) to measure a mutant's fitness. The role of the fitness function in GenProg is to guide the search process through program mutations, retaining those that encode the desired behavior. Ideally, a set of code changes can be found that result in a version of the program that passes all relevant test cases and thus represents a valid patch for the associated bug. Conceptually, counting passing tests is a simple method for quantifying how much correct behavior a given mutant exhibits and could thus guide an efficient search strategy. In practice, however, GenProg can take a long time to find even small, single-edit repairs [47], seemingly trying random mutations until it happens upon the one that fixes the bug. Thus, we desire a better understanding of the current fitness signal to motivate and direct a better fitness function to improve automated program repair.

4.2.2 An evaluation of GenProg's fitness function

While previous work has examined the possibility of improving fitness functions to speed up the repair process and elicit more fixes in practice [175, 176, 177, 178], there are still many bugs that GenProg fails to fix in a practical amount of time given ample resources [174]. Our approach to measuring fitness is heuristic in nature and thus we focus on two fundamental goals: speed and accuracy.

In generate-and-validate program repair approaches, the majority of the computation cost lies in validation (i.e., evaluating the fitness function on potential patches). This motivates the need for fast fitness evaluations. The speed of GenProg's fitness function has been addressed previously by sampling test suites as a heuristic approximation for full test suite evaluation [176]. We investigate test prioritization strategies later in this chapter to further speed up the speed of our framework's fitness function.

We employ a heuristic approach to measuring program variant fitness and, as such, the accuracy of the underlying approximation is important when trying to optimize the search for valid patches. At a high level, we want to minimize the computational cost of running our patch generation algorithm. This overall metric is difficult to model precisely, so we instead focus on fitness accuracy, which can be measured more directly given GenProg's artifacts and also affects the efficiency and thus cost of the underlying search for patches. One way to measure fitness accuracy is fitness distance correlation (FDC) [179]. Fitness distance correlation measures whether a fitness signal accurately models

some ground-truth notion of how close a mutant is to the desired goal. In the case of GenProg, a good fitness function should output optimal values for mutants that are conceptually closer to an actual patch. Put another way, assuming we know the set of necessary program changes that constitute a valid patch for a given defect, fitness values should be positively correlated with the fraction those changes a given mutant contains.

We use historical GenProg data to approximate the set of changes associated with valid fixes to certain bugs. We then measure how close a given mutant is to an eventual fix by looking at the intersection of its mutations and those in an eventual fix. For instance, if we know mutations W, X, Y, and Z constitute a patch for a given bug, then a mutant A that comprises changes X, Y, and Z is quantitatively closer to a fix than mutant B that only comprises mutations W and X. If we measure "closeness to a fix" in this manner, we can use it as a ground-truth notion of fitness to gauge the accuracy of GenProg's measured fitness heuristic. By correlating computed fitness with this notion of actual fitness we can calculate GenProg's FDC. Using data collected from 20 bugs fixed in experiments published in 2012 [47], we calculated the fitness-distance correlation of the `GenProg` tool to be 0.145 (values between -0.15 and 0.15 are commonly considered "uncorrelated" [179]). This suggests that measuring the number of test cases a program variant passes alone does not adequately measure how close a given set of mutations is to an eventual fix. We thus desire a fitness function that more closely encodes desired program behavior and can thus potentially better direct the search for defect patches.

4.2.3 Investigations into GenProg's fitness function

Program behavior is the means by which we determine if a candidate patch fixes the bug in question. Thus, we maintain a testing-based fitness function because testing directly measures program behavior, but we considered several changes to increase fitness distance correlation. The existing model of variant fitness in the GenProg framework assumes that all test cases are equally representative of the desired program behavior. In a pilot study, we investigated the validity of this assumption by examining the mutants created when fixing 10 representative bugs [47]. One way to measure the utility of a test case in the context of our program repair framework is to measure the correlation between its outcome and actual fitness (described in the previous subsection). Using 6,675 mutants created by GenProg as a part of the bug fixing process, we found significant variation in different test cases' correlations (using the Pearson product moment coefficient). Specifically, some test cases exhibit over 4.5 time more correlation with edits indicative of a future fix than others. This would suggest that the state-of-the-art scheme of assigning all test cases equal weights when computing fitness may not be the optimal method.

One way to find a beneficial test case weighting scheme is to measure the relative importance of various test cases using the actual fitness of a given variant (i.e., its actual, post hoc fitness, measured by what percentage of eventually relevant program mutations it contains). Any test case that is passed by variants with high actual fitness but failed by lower actual fitness variants could then be weighted highly in a fitness function because of its high discriminatory power.

We evaluated several machine learning algorithms (e.g., Naïve Bayes classification or multilayer perceptron neural networks) to maximize the weighting's effectiveness. Unfortunately, none of the models we tried yielded a statistically significant increase in fitness distance correlation. That is, no simple model based on test case weightings was found to be a good signal to drive the search for program repairs.

We considered FDC as a proxy for the computational cost of the algorithm (see Section 4.2.2), but found no method for significantly increasing this metric in practice. We thus focus on concrete cost metrics (e.g., the number of mutants evaluated and the wall-clock running time) as they more closely align with our ultimate goal: cost savings. Additionally, in further pilot studies we examined the effects of varying several of GenProg's parameters including the test suite sampling rate, generation size, fault localization weights, and the random seeds used. None of these changes had a favorable effect on the associated cost metrics on average, including the number of trials that produce patches when the algorithm used different seeds. One possibility for explaining these results is that our experimental design was flawed or that we were not considering the right parameters.

Ultimately, however, research intuition suggested that a completely redesigned framework might yield more significant cost savings than small modifications to the existing approach. Put another way, the results presented in this section suggest that small incremental changes are ineffective at improving GenProg and that more sweeping, fundamental changes may be necessary to generate more patches, more quickly.

4.2.4 Investigating historical bug fixes and previously unpatched bugs

To motivate and explore fundamental changes to the GenProg algorithm, we investigated both existing bug fixes to expose possible speed-ups and also previously unpatched bugs to suggest additional fix strategies.

We evaluated previous repairs (specifically, the search strategy) to identify bottlenecks that might be alleviated to reduce the cost of the algorithm. When a valid patch is found, GenProg uses a minimization technique (very similar to delta debugging [126]) to output the minimal set of program changes that constitute a valid patch for the given defect. For the 55 bugs GenProg fixes in previous work [47], we found a total of 716 unique, minimized, validated patches in all previous experiments (notably, a single bug can often be repaired in more than one way — we found an average of 13 repairs per bug). Of these 716 patches, 710 included only a single edit while the remaining six contained only two edits each. GenProg was designed to produce high-order (i.e., multiple-edit) patches by construction: each generation combines previous sets of mutations and adds an additional one [129]. While it was hypothesized that complex program changes might be needed to fix bugs, these results suggest that in practice, higher-order bug fixes are not produced by GenProg.

Figure 4.1 shows the number of generations (or new edits) GenProg took to find a patch on average for the

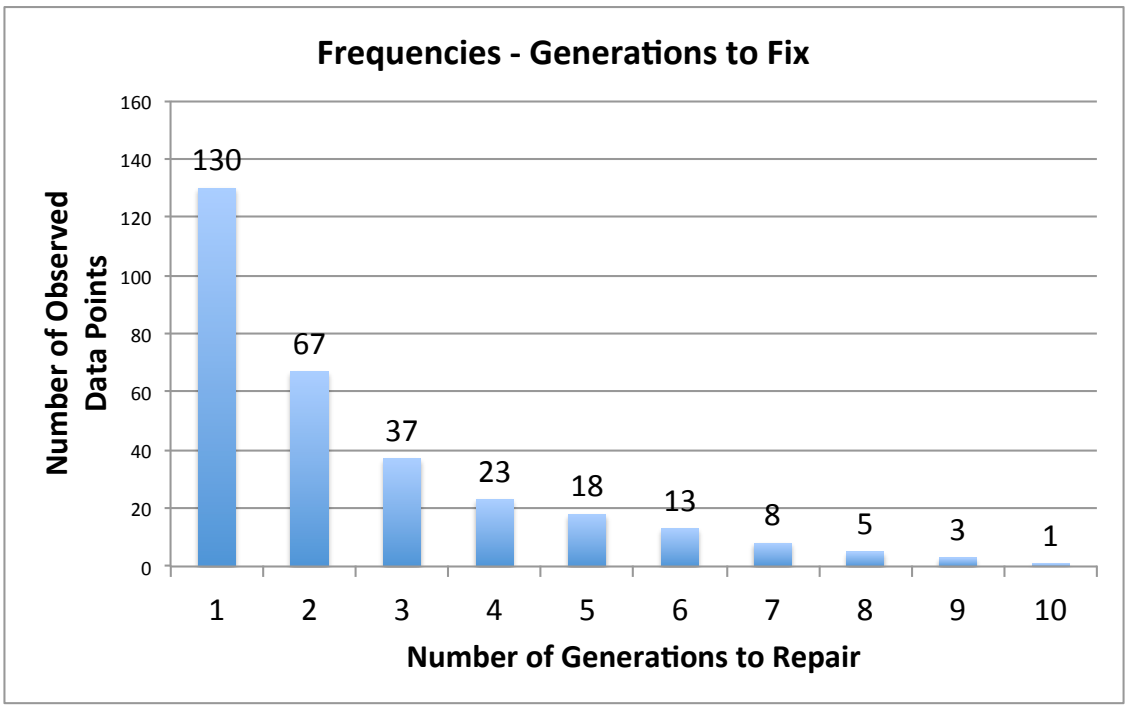

Figure 4.1: The graph shows the distribution of the number of generations it took to find the 305 unique bug fixes for the 55 bugs fixed in previous work [47].

experiments presented in [47].[1] While many patches are found early (e.g., within one or two generations), the graph exhibits a long tail. This tail, combined with the fact that the great majority of found patches involve only a single edit, suggests that the current search strategy is sub-optimal. For this data set, GenProg reaches a patch after an average of 2.6 generations, while optimally only one generation is needed for the majority of the associated single-edit fixes. Previous work on "software mutational robustness" found that it is possible to make a substantial number of program changes without changing its tested behavior [180]. This suggests that GenProg might often take "random walks" throughout the search space (i.e., making mutations that do not positively or negatively affect program behavior) and happen upon a fix after several unsuccessful mutations.

The findings from the previous study on fitness distance correlation (Section 4.2.3) are less surprising when we consider that most fixes consist of a single mutation. Put another way, fitness distance correlation assumes that an ideal search strategy "builds" a good solution piece-by-piece; the state-of-art strategy appears to be akin to an "all or nothing" approach which admits little to no granularity of measurement. While previous evidence suggests we may not be able to improve GenProg's framework, these results suggest that we may be able to focus specifically on finding single-edit repairs quickly to reduce the cost of patch generation for, on average, half of the bugs we have studied.

As a complimentary study, we manually examined the bugs GenProg historically failed to fix from the largest

[1]Note: this data set contains 305 fixes that were all generated using the same experimental framework found in Le Goues *et al.* [47]. The 716 fixes mentioned previously were taken from several incomparable experimental setups and thus cannot all be used to investigate the number of generations needed to generate patches.

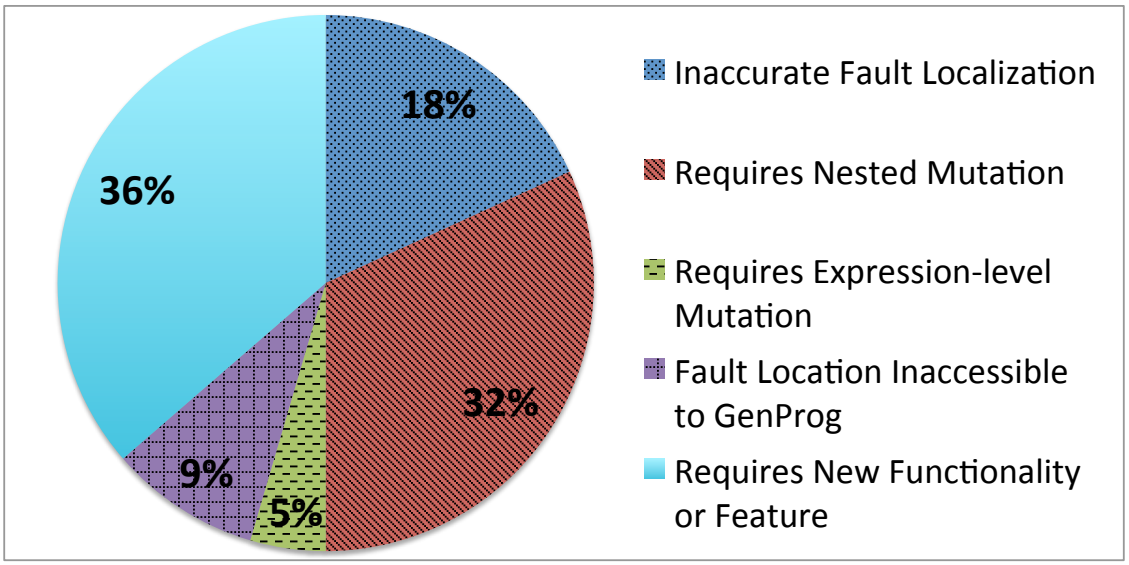

Figure 4.2: The pie graph shows the distribution of causes for bugs unpatched by GenProg. We examined historical human-written fixes for each bug in question and categorized each based on the current assumptions and limitations of GenProg.

previously used data set [47]. Notably, we hypothesize that there is no single reason for GenProg's failure to patch additional bugs and thus we examine the causes of failure given the state-of-the-art genetic algorithm-based framework and experimental parameters. Figure 4.2 shows the range and distribution of limitations that likely cause GenProg's failure to find a patch for 50 bugs from this historical data set. We systematically examined the human patch for each bug to better understand the nature of the corresponding defect and classified the likely cause based on GenProg's limitations. In certain cases, such as "inaccurate fault localization," finding a valid repair is not impossible, but is statistically unlikely, even if the algorithm is run for a long time. Inaccurate fault localization information may make the search impractical and cause it to favor mutations in parts of the code unrelated to the underlying problem. However, the other four categories, which account for 82% of the remaining bugs, are outside of the scope of traditional GenProg in terms of expressive power. For instance, the state-of-the-art implementation of GenProg does not consider nested mutations by default (i.e., mutating previous mutated statements). In fact, as mentioned in the previous paragraph, GenProg rarely finds even two-edit fixes in practice, suggesting that the size of the multi-edit search space is generally too large to make nested mutations practical in the current framework.

These results suggest that targeting new fixes for previously unfixed bugs would require large, sweeping changes to the framework. Furthermore, there is little evidence in favor of such incremental techniques and, in fact, evidence suggesting they may not be independently successful, which makes them higher-risk research strategies. Taking into consideration the insights presented in this section, we focus on reducing the cost for those bugs that GenProg can already repair rather than generating new patches for previously unfixed bugs. Subsequent sections outline a new, more accurate, cost model and an updated approach to generating single-edit patches.

4.3 Motivating a new search strategy

An early and simplistic cost model for GenProg related the number of complete test suite evaluations to the size of the parts of the program implicated by fault localization [181, Fig. 9]. This is intuitive, but incomplete because it ignores the test-suite sampling discussed earlier, and it ignores the order in which candidate repairs are evaluated (e.g., if a high-probability candidate were validated early, the search could terminate immediately, reducing the incurred cost) and the number of possible repair operations (edits) that can be considered. For example, the single largest performance optimization for GenProg to date, the use of test suite sampling or reduction for internal fitness value calculations (retaining the use of the entire suite to validate candidate repairs), improves performance by 80% overall in terms of test suite evaluations [173, Fig. 2], but is invisible to a cost model defined solely in terms of fault localization. Similarly, the order in which repairs are enumerated and the number of implicated fixes (not just faulty lines) clearly influence the total cost but are not considered as factors. However, GenProg demonstrates high variability, both across individual trials and among programs and defects. For example, in one large study, GenProg's measured ability to repair a defect varied from 0–100% with no clear explanation [174, Fig. 4]. In light of such results, a more powerful explanatory framework is desired.

A second cost arises from syntactically distinct but semantically equivalent program variants. This overhead is real [88, 89] but completely ignored by cost models that consider only test case evaluations. In a generate-and-validate repair framework, equivalent programs necessarily have equivalent behavior on test cases, so the size of this effect can be estimated by considering the number of candidate repairs that have exactly the same test case behavior. To this end, we examined the test output of over 500,000 program variants produced by GenProg in a bug repair experiment [47]. For a given bug, if we group variants based on their test output, 99% of them are redundant with respect to tested program behavior, on average. Although not all programs that test equally are semantically equivalent, this suggests the possibility of optimizing the search by recognizing and avoiding redundant evaluations.

We thus desire a more descriptive cost model as well as a search-based repair algorithm that explicitly considers program equivalence and the order in which tests and edits are explored.

4.4 Cost Model

This section outlines a cost model for generate-and-validate repair to guide the algorithmic improvements outlined in Section 5.3. We assume a repair algorithm that generates and validates candidate repairs using test cases, and tests are the dominant cost. We acknowledge many differences between approaches but believe the description in this section is sufficiently general to encompass techniques such as GenProg [47], ClearView [39], Debroy and Wong [138], and PAR [58].

Broadly, the costs in a generate-and-validate algorithm depend on generation (how many candidates are created) and validation (how each one is tested). Without optimization, the number of tests executed equals the number of candidate repairs considered by the algorithm multiplied by the size of the test suite. Fault localization typically identifies a region of code that is likely associated with the bug. Fault localization size refers to the number of statements, lines of code, or other representational unit manipulated by the repair algorithm. A candidate repair (i.e., a patch) typically modifies only the code identified by fault localization, but it can also include code imported from another part of the program [47], synthesized [137] or instantiated from a template [39, 58]. The number of first-order candidate repairs is the product of the fault localization size and the size of the *fix space* [47, Sec. V.B.3], where fix space refers to the atomic modifications that the algorithm can choose from. In addition, the repair algorithm could terminate when it finds and validates a repair, so the enumeration strategy—the order in which candidate repairs are considered—can have significant impact. Similarly, a non-repair may be ruled out the first time it fails a test case, and thus the testing strategy also has a significant impact.

4.1 shows our cost model. Fault localization size is denoted by Fault. The number of possible edits (mutations) is denoted by Fix. Note that the model structure has each component depending on the previous components. For example, Fix depends on Fault (e.g., some templates or edit actions might not apply depending on the variables in scope, control flow, etc.). The size of the test suite is denoted by Suite. The order in which the algorithm considers candidate repairs is $\mathsf{RepairStrat}$, and $\mathsf{RepairStratCost}$ denotes the number of tests evaluated by $\mathsf{RepairStrat}$, which ranges from $1/(\mathsf{Fault} \times \mathsf{Fix})$ (an optimal strategy that selects the correct repair on the first try) to 1 (a pessimal strategy that considers every candidate). Finally, given a candidate repair, the order in which test cases are presented is given by $\mathsf{TestStrat}$, and the number of tests evaluated by $\mathsf{TestStrat}$ is given by $\mathsf{TestStratCost}$, which ranges from $1/\mathsf{Suite}$ (optimal) to 1 (worst case).

$$\begin{aligned} \mathsf{Cost} &= \mathsf{Fault} \times \mathsf{Fix}(\mathsf{Fault}) \times \mathsf{Suite}(\mathsf{Fault}, \mathsf{Fix}) \\ &\times \mathsf{RepairStratCost}(\mathsf{Fault}, \mathsf{Fix}, \mathsf{Suite}) \\ &\times \mathsf{TestStratCost}(\mathsf{Fault}, \mathsf{Fix}, \mathsf{Suite}, \mathsf{RepairStratCost}) \end{aligned} \tag{4.1}$$

By contrast, earlier algorithms defined the search space as the product of Fault and Fix for a given mutation type [174, Sec. 3.4]. GenProg's policy of copying existing program code instead of inventing new text corresponds to setting Fix equal to Fault (leveraging existing developer expertise and assuming that the program contains the seeds of its own repair) for $\mathcal{O}(N^2)$ edits, while other techniques craft repairs from lists of templates [39, 58, 130]. Although fault localization is well-established, *fix localization*, identifying code or templates to be used in a repair, is just beginning to receive attention [58].

In our model, Fix depends on Fault, capturing the possibility that operations can be avoided that produce ill-typed

programs [135] or the insertion of dead code [88, 89]. Suite depends on Fix so the model can account for techniques such as impact analysis [182]. The RepairStrat term expresses the fact that the search heuristic ultimately considers candidate repairs in a particular order, and it suggests one way to measure optimality. It also exposes the inefficiencies of algorithms that re-evaluate semantically equivalent candidates. The TestStrat term depends on the repair strategy, allowing us to account for savings achieved by explicit reasoning about test suite sampling [173, 181]. Note that Suite optimizations remove a test from consideration entirely while TestStrat optimizations choose remaining tests in an advantageous order.

In the next section, we use the structure of the cost model, with a particular emphasis on the repair and test strategy terms, to outline a new search-based repair algorithm.

4.5 Repair Algorithm

We introduce a novel automated program repair algorithm motivated by the cost model described in Section 4.4. Based on the observation that running test cases on candidate repairs is time-consuming, the algorithm reduces this cost using several approaches. First, it uses an approximate program equivalence relation to identify candidate repairs that are semantically equivalent but syntactically distinct. Next, it controls the order in which candidate repairs are considered through an adaptive search strategy. A second adaptive search strategy presents test cases to candidate repairs intelligently, e.g., presenting test cases early that are most likely to fail. Although each of these components adds an upfront cost, our experimental results show that we achieve net gains in overall time performance through these optimizations. To highlight its use of *A*daptive search strategies and program *E*quivalence, we refer to this algorithm as "AE" in this chapter.

We first describe the algorithm and then provide details on its most important features: the approximate program equivalence relation and two adaptive search strategies.

4.5.1 High-level description

The high-level pseudocode for our algorithm is given in Figure 4.3. It takes as input a program P, a test suite Suite that encodes all program requirements and impact analyses, a conservative approximate program equivalence relation $\sim$, an edit degree parameter k, an edit operator Edits that returns all programs resulting from the application of kth order edits, and the two adaptive search strategies RepairStrat and TestStrat. The algorithm is shown enumerating all kth-order edits of P on line 3. In practice, this is infeasible for $k > 1$, and operations involving $CandidateRepairs$ should be performed using lazy evaluation and calculated on-demand. On line 5 the RepairStrat picks the candidate repair deemed most likely to pass all tests based on $Model$, the observations thus far. On lines 8 and 9 the quotient space is computed lazily: A set of equivalence classes encountered thus far is maintained, and each new candidate

Input: Program $P : \text{Prog}$
Input: Test suite $\mathsf{Suite} : \text{Prog} \to \mathcal{P}(\text{Test})$
Input: Equivalence relation $\sim : \text{Prog} \times \text{Prog} \to \mathcal{B}$
Input: Edit degree parameter $k : \mathcal{N}$
Input: Edit operator $\mathsf{Edits} : \text{Prog} \times \mathcal{N} \to \mathcal{P}(\text{Prog})$
Input: Repair strategy $\mathsf{RepairStrat} : \mathcal{P}(\text{Prog}) \times \text{Model} \to \text{Prog}$
Input: Test strategy $\mathsf{TestStrat} : \mathcal{P}(\text{Test}) \times \text{Model} \to \text{Test}$
Output: Program P'. $\forall t \in \mathsf{Suite}(P').\ P'(t) = \textbf{true}$

```
 1: let Model ← ∅
 2: let EquivClasses ← ∅
 3: let CandidateRepairs ← Edits(P, k)
 4: repeat
 5:    let P′ ← RepairStrat(CandidateRepairs, Model)
 6:    CandidateRepairs ← CandidateRepairs \ {P′}
 7:    // "Is any previously tried repair equivalent to P′?"
 8:    if ¬∃ Previous ∈ EquivClasses. P′ ∼ Previous then
 9:       EquivClasses ← EquivClasses ∪ {P′}
10:       let TestsRemaining ← Suite(P′)
11:       let TestResult ← true
12:       repeat
13:          let t ← TestStrat(TestsRemaining, Model)
14:          TestsRemaining ← TestsRemaining \ {t}
15:          TestResult ← P′(t)
16:          Model ← Model ∪ {⟨P′, t, TestResult⟩}
17:       until TestsRemaining = ∅ ∨ ¬TestResult
18:       if TestResult then
19:          return P′
20:       end if
21:    end if
22: until CandidateRepairs = ∅
23: return "no k-degree repair"
```

Figure 4.3: Pseudocode for adaptive equivalence ("AE") generate-and-validate program repair algorithm. Candidate repairs P' are considered in an order determined by RepairStrat, which depend on a $Model$ of observations ($Model$ may be updated dynamically while the algorithm is running) and returns the edit (mutation) deemed most likely to pass all tests. Candidate repairs are compared to previous candidates and evaluated only if they are not semantically equal with respect to an approximate program equivalence relation ($\sim$). TestStrat determines the order in which tests are presented to P', returning the test on which P' is deemed most likely to fail. The first P' to pass all tests is returned.

repair is checked for equivalence against a representative of each class. If the new candidate is equivalent to a previous one, it is skipped. Otherwise, it is added to the set of equivalence classes (line 9). Candidates are evaluated on the relevant test cases (line 10) in an order determined by TestStrat (line 13), which uses information observed thus far to select the test deemed most likely to fail. Since successful repairs are run on all relevant tests regardless, TestStrat affects performance (opting out after the first failure) rather than functionality, and is thus chosen to short-circuit the loop (lines 12–17) as quickly as possibly for non-repairs. If all tests pass, that candidate is returned as the repair. If all semantically distinct candidates have been tested unsuccessfully, there is no k-degree repair given that program, approximate equivalence relation, test suite and set of mutation operators.

The cost model identifies five important components: Fault, Fix, Suite, RepairStrat, and TestStrat. We leave fault localization (Fault) as an orthogonal concern [37], although there is some recent interest in fault localization

targeting automated program repair rather than human developers [183]. In this dissertation, we use the same fault localization scheme as GenProg [129] to control for that factor in our experiments. Similarly, while we consider impact analysis [182] to be the primary Suite reduction, we do not perform any such analysis in this dissertation to admit a controlled comparison to GenProg, which also does not use any. Finally, one cost associated with testing is compiling candidate repairs; compilation costs are amortized by bundling multiple candidates into one executable, each selected by an environment variable [184]. In the rest of this section we discuss the other three components.

4.5.2 Determining Semantic Equivalence

To admit a direct, controlled comparison we form Edits as a quotient space of edits produced by the GenProg mutation operators "*delete* a potentially faulty statement" and "*insert* after a potentially faulty statement a statement from elsewhere in the program." This means that any changes to the search space of edits are attributable to our equivalence strategies, not to different atomic edits or templates. GenProg also includes a "*replace*" operator that we view as a second-degree edit (delete followed by insert); in this dissertation we use edit degree $k = 1$ unless otherwise noted (see Section 4.6 for a further examination of degree).

If two deterministic programs are semantically equivalent they will necessarily have the same test case behavior.[2] Thus, when we can determine that two different edits applied at the same fault location would yield provably equivalent programs, the algorithm considers only one. Since general program equivalence is undecideable, we use a sound approximation $\sim$: $A \sim B$ implies that A and B are semantically equivalent, but our algorithm is not guaranteed to find all such equivalences. We can hope to approximate this difficult problem because we are not dealing with arbitrary programs A and B, but instead A and A', where we constructed A' from A via a finite sequence of edits applied to certain locations. Although our algorithm is written so that the quotient space is computed lazily, for small values of k it can be more efficient to compute the quotient space eagerly (i.e., on line 3 of Figure 4.3).

In this domain, the cost of an imprecise approximation is simply the additional cost considering redundant candidate repairs. This is in contrast with mutation testing, where the equivalent mutant problem can influence the quality of the result (via its influence on the mutation score, see Section 4.7). Drawing inspiration from such work, we determine semantic equivalence in three ways: syntactic equality, dead code elimination, and instruction scheduling.

Syntactic Equality Programs often contain duplicated variable names or statements. In techniques like GenProg that use the existing program as the source of insertions, duplicate statements in the existing program yield duplicate insertions. For example, if the statement `x=0` appears k times in the program, GenProg might consider k separate edits, inserting each instance of `x=0` after every implicated fault location. Template-based approaches are similarly influenced:

[2] Excluding non-functional requirements, such as execution time or memory use. We view such non-functional program properties as a separate issue (i.e., compiler optimization).

if `ptr` is both a local and a global variable and a null-check template is available, the template can be instantiated with either variable, leading to syntactically identical programs. Programs that are syntactically equal are also semantically equal, so $A =_{text} B \implies A \sim B$.

Dead Code Elimination If `lval` is not live at a proposed point of insertion, then a write to it will have no effect on program execution (assuming `rval` has no side-effects [100]). If k edits $e_1 \dots e_k$ applied to the program A yield a candidate repair $A[e_1 \dots e_k]$ and e_i inserts dead code, then $A[e_1 \dots e_k] \sim A[e_1 \dots e_{i-1} e_{i+1} \dots e_k]$. As a special common case, if e_1 inserts dead code then $A[e_1] \sim A$. Dataflow analysis allows us to determine liveness in polynomial time, thus ruling out insertions that will have no semantic effect.

Instruction Scheduling Consider the program fragment `L1: x=1; L2: y=2; L3: z=3;` and the Fix mutation "insert `a=0;`." Our algorithm would consider three possible insertions: one at `L1`, one at `L2` and one at `L3`. In practice, all three insertions are equivalent: `a=0` does not share any read-write or write-write dependencies with any of those three statements. More generally, if `S1; S2;` and `S2; S1;` are semantically equivalent, only one of them need be validated. One type of instruction scheduling compiler optimization moves (or "bubbles") independent instructions past each other to mask latencies or otherwise improve performance. We use a similar approach to identify this class of equivalences quickly.

First, we calculate effect sets for the inserted code and the target code statements (e.g., reads and writes to variables, memory, system calls, etc.). If two adjacent instructions reference no common resources (or if both references are reads), reordering them produces a semantically equivalent program. If two collocated edits e and e' can be instruction scheduled past each other, then $A[\dots ee' \dots] \sim A[\dots e'e \dots]$ for all candidate repairs A. This analysis runs in polynomial time.

Precision in real applications typically requires a pointer alias analysis (e.g., must `*ptr=0` write to `lval` and/or may `*ptr` read from `lval`). For the experiments in this dissertation, we implement our flow-sensitive, intraprocedural analyses atop the alias and dataflow analysis framework in CIL [100].

4.5.3 Adaptive Search Strategies

The repair enumeration loop iterates until it has considered all atomic edits, stopping only when (and if) it finds one that passes all tests. Similarly, the test enumeration loop iterates through all the tests, terminating only when it finds a failing test or has successfully tested the entire set. In this section we discuss our algorithmic enhancements to short-circuit these loops, improving performance without changing semantics.

There are many possible strategies for minimizing the number of interactions in both loops. For the experiments in this chapter we use a simple, non-adaptive RepairStrat as a control: as in GenProg, edits are preferred based on their

fault localization suspiciousness value. By contrast, for TestStrat we favor the test that has the highest historical chance of failure (in the $Model$), breaking ties in favor of the number of times the test has failed and then in favor of minimizing the number of times it has passed. Although clearly a simplification, these selection strategies use information that is easy to measure empirically and are deterministic, eliminating algorithm-level variance.

Although these strategies are quite simple, they are surprisingly effective (Section 4.6). However, we expect that future work will consider additional factors, e.g., the running time of different test cases, and could employ machine learning or evolutionary algorithms to tune the exact function.

4.6 Experiments

We present experimental results evaluating *AE* described in Section 5.3, using the ICSE 2012 dataset [47] as our basis of comparison. We focus on the following issues in our experiments:

1. Effectiveness at finding repairs: How many repairs from [47] are also found by AE?
2. Search-space efficiency: How many fewer edits are considered by AE than GenProg?
3. Cost comparison: What is the overall cost reduction (test case evaluations, monetary) of AE?
4. Optimality: How close is AE to an optimal search algorithm? (Section 4.5.3)?
5. Generality: What is the range of bug types AE fixes in practice?

4.6.1 Experimental Design

Our experiments are designed for direct comparison to previous GenProg results [47], and we use the publicly available benchmark programs from this earlier work. This data set contains 105 high-priority defects in eight programs totaling over 5MLOC and guarded by over 10,000 tests.

We provide grounded, reproducible measurements of time and monetary costs via Amazon's public cloud computing services. To control for changing prices, all values reported here use the same Aug-Sep 2011 prices [47, Sec. IV.C] unless otherwise noted. These GenProg results involved ten random trials run in parallel for at most 12 hours each (120 hours per bug). Since AE is deterministic, we evaluate it with a single run for each bug, allowing for up to 60 hours per bug.

4.6.2 Success Rates, Edit Order, Search-space Size

Table 4.1 shows the results. k indicates the maximum possible number of allowed mutations (edits). Ignoring the crossover operator, a 10-generation run of GenProg could in principle produce an individual with up to 10 mutations.

			Order 1 Search Space		Defects Repaired			Test Suite Evals.		US$ (2011)	
			AE	GP	AE	GP	GP	AE	GP	AE	GP
Program	LOC	Tests	$k = 1$	$k = 1$	$k = 1$	$k = 1$	$k \leq 10$	$k = 1$	$k \leq 10$	$k = 1$	$k \leq 10$
fbc	97,000	773	507	1568	1	0	1	1.7	1952.7	0.01	5.40
gmp	145,000	146	9090	40060	1	0	1	63.3	119.3	0.91	0.44
gzip	491,000	12	11741	98139	2	1	1	1.7	180.0	0.01	0.30
libtiff	77,000	78	18094	125328	17	13	17	3.0	28.5	0.03	0.03
lighttpd	62,000	295	15618	68856	4	3	5	11.1	60.9	0.03	0.04
php	1,046,000	8,471	26221	264999	22	18	28	1.1	12.5	0.14	0.39
python	407,000	355	—	—	2	1	1	—	—	–	—
wireshark	2,814,000	63	6663	53321	4	1	1	1.9	22.6	0.04	0.17
weighted sum	—	—	922,492	7,899,073	53	37	55	186.0	3252.7	4.40	14.78

Table 4.1: Comparison of AE and GenProg on successful repairs. AE denotes the "adaptive search, program equivalence" algorithm described in Figure 4.3 and columns labeled GP reproduce previously published GenProg results [47]. k denotes the maximum number of edits allowed in a repair (note: in the case of GenProg, this is the number of generations explored — the number of edits could feasibly be greater than the number of generations based on crossover techniques). Additionally, edits are performed on a program's abstract syntax tree nodes and thus a single edit may constitute a considerable amount of changed statements if the associated node has many children. The first three columns characterize the benchmark set. The "Search Space" columns measure number of first-order edits considered by each method in the worst case. The "Defects Repaired" columns list the number of valid patches found: only 45 defects are repaired by both algorithms (e.g., there are no shared repairs for **python**). The "Test Suite Evals." column measures the average number of test suite evaluations on those 45 repaired defects. The monetary cost column measures the average public cost of using Amazon's cloud computing infrastructure to to find those 45 repairs.

However, this is an extremely rare event, because of the small population sizes used in these results (40) and the effects of finite sampling combined with selection.

The "Defects Repaired" column shows that AE, with $k = 1$ and restricted to 60 CPU-hours, finds repairs for 48 of the 105 original defects. GenProg, with $k \leq 10$ and 120 CPU-hours, repairs 55. This confirms earlier GenProg results using minimization that show a high percentage (but not all) of the bugs that GenProg can repair can also be repaired with one or two edits. As a baseline to show that this result is not specific to AE, we also consider a version of GenProg restricted to one generation ($k = 1$): it finds 37 repairs.

The remaining experiments include just the 45 defects that both algorithms repair, allowing direct comparison. The "Search Space" column measures the number of possible first-order edits. Higher-order edits are too numerous to count in practice in this domain: first-order insert operations alone are $\mathcal{O}(n^2)$ in the size of the program, and 10 inserts yields $\mathcal{O}(n^{20})$ options. The results show that using program equivalence (Section 5.3) dramatically reduces the search space by 87.5%, when compared with GenProg.

4.6.3 Cost

Since neither algorithm is based on purely random selection, reducing the search space by x does not directly reduce the expected repair cost by x. We thus turn to two externally visible cost metrics: test suite evaluations and monetary cost.

Test suite evaluations measure algorithmic efficiency independent of systems programming or implementation details. The "Test Suite Evals." column reports shows that AE requires order-of-magnitude fewer test suite evaluations

than does GenProg: 186 vs. 3252. Two factors contribute to this fifty-fold decrease: search-space reduction and test selection strategy (see Section 4.6.4).

Finally, we ground our results in US dollars using public cloud computing. To avoid conflating our improvements with Amazon price reductions, we use the applicable rate used in the earlier GenProg evaluation ($0.074 dollars per CPU-hour, including data and I/O costs). For example, on the `fbc` bug, serial AE algorithm runs for 0.14 hours and thus costs $0.14 \times 0.074 =$ one cent. GenProg runs ten machines in parallel, stopping when the first finds a repair after 7.29 hours, and thus costs $7.29.52 \times 10 \times 0.074 = \$5.40$. Overall, AE is cheaper than GenProg by a factor of three ($4.40 vs. $14.78 for 45 successful repairs achievable by both algorithms).

4.6.4 Optimality

The dramatic decrease in the number of test suite evaluations performed by AE, and the associated performance improvements, can be investigated using our cost model. Our experiments used a simple repair strategy (fault localization) and a dynamically adjusted (adaptive) test strategy. In the cost model, TestStrat depends on RepairStrat: Given a candidate repair, the test strategy determines the next test to apply. For a successful repair, in which n candidate repairs are considered, an *optimal* test strategy would evaluate $(n-1) + |\mathsf{Suite}|$ test cases. In the ideal case, the first $n-1$ candidate repairs would each be ruled out be a single test and the ultimate repair would be validated on the entire suite.

We now measure how close the technique described in Section 5.3 approaches this optimal solution in practice. For example, for the 20 `php` shared instances, GenProg runs 1,918,170 test cases to validate 700 candidate repairs. If the average test suite size is 7,671, an optimal test selection algorithm would run $680 + 20 \times 7,671 = 154,100$ test cases. GenProg's test selection strategy (random sampling for internal calculations followed by full evaluations for promising candidates [173]) is thus $12\times$ worse than optimal on those bugs. On those same bugs, AE runs 163,274 test cases to validate $3,099$ mutants. Its adaptive test selection strategy is thus very near optimal, with a $0.06\times$ increase in testing overhead on those bugs. By contrast, the naïve repair selection strategy evaluated is worse than GenProg's tiered use of fault localization, mutation preference, fix localization, and past fitness values. Despite evaluating $4\times$ times as many candidates, however, we evaluate $12\times$ fewer tests. The results on other programs are similar.

This analysis suggests that integrating AE's test selection strategy with GenProg's repair selection strategy, or enhancing AE's adaptive repair strategy, could lead to even further improvements. We leave the exploration of these questions for future work.

The difference between our test count reduction and our monetary reduction stems from unequal test running times (e.g., AE selects tests with high explanatory power but also above-average running times).

Category	Subcategory	Explanation
User interface	Functionality	The UI implementation is awkward, incomplete, or incorrect)
	Output	The output (i.e. displayed results) of a program is correct
Error Handling	Recovery	Inability to gracefully recover from exceptional behavior
Boundary errors	Numeric	Out-of-bounds access to data structures
	Loops	Incorrect iteration or termination conditions
Calculation errors	Ordering	Incorrect order of expressions or operations
	Incorrect Algorithm	Use of the wrong formula or algorithm for the problem at hand
Initial states	Numeric	Failure to initialize or reset a loop control variable
	String	Failure to set or clear a string variable
Control Flow	Conditional	Wrong conditional used to control branching
Data	Handling	Mistakenly corrupting or incorrectly packaging/casting persistent data
	Interpreting	Incorrectly understanding the type, format, or nature of program data
	Indexing	Wrong field in a table or mask for a bit field
Race Conditions	Recognition	Incorrect assumption about the order or dependency of events

Table 4.2: This table presents a subset of the defect taxonomy proposed by Kaner, Falk, and Nguyen [53]. For our purposes, we restrict the taxonomy to categories related specifically to coding errors.

4.6.5 Generality

One of the overarching goals of this dissertation is generality (i.e., the notion that a more widely-applicable tool may have greater possible impact with respect to software maintenance cost savings). This subsection measures AE's generality in terms of the types of defects for which it could potentially generate patches. One way to measure generality is to count how many different types of bugs a given tool fixes in practice. However, different tools may patch the same bug in different ways, making the classification process ambiguous — classifying the types of patches may lead to conflicting conclusions. For instance, AE's generated patches often differ from those created by humans when attempting to fix the same bugs. Additionally, we lack the experimental data sets necessary to directly reason about the generality of competing state-of-the-art techniques to put this work in context. We thus focus abstractly on a given technique's expressive power and argue which types of bugs it could fix in a best case scenario (e.g., optimal search and sufficient time). We adapt the defect taxonomy of Kaner, Falk, and Nguyen [53] by considering only those categories of defects that correspond to errors in the code; AE can only affect a system's code and thus we consider only defect classes related to the code itself. For example their original taxonomy contains errors related to requirements gathering which is out of the scope of this work. The defect classes that meet these criteria are described in Table 4.2. We acknowledge that this taxonomy may not include every defect type but feel that it is representative of many types in practice.

If we consider the expressive power of GenProg and AE by focusing on the mutation operators inherent in each technique, we assert that it is generically possible (i.e., given an optimal search strategy and sufficient time) to fix

examples of each of the bug types outlined in Table 4.2. This does not, however, imply that GenProg and AE *will* fix all of the associated bug types every time, in practice. Variances in search space, bug complexity, and search strategy coupled with limited time and computing resources translates to fluctuation in the techniques' success rates. Our theoretical conclusions about expressive power are empirically grounded in Section 4.6.2, which presents evidence that AE and GenProg fix largely the same number of bugs in practice.

Section 2.4.2 outlines many state-of-the-art repair techniques. AE can theoretically fix strictly more bug types than certain narrowly focused existing approaches (e.g., AFix that targets only single-variable atomicity violations [26]). At worst, AE's expressive power is no worse than that of existing techniques, with respect to fixing a wide range of bug types (i.e., it is comparable to ClearView [39], AutoFix-E [130], and Debroy and Wong [138]). ClearView makes directed, but widely-focused (with respect to different types of bugs), program changes to binaries to alter and ideally correct failing program invariants; AE can similarly affect the associated invariants although it does not consider them directly. AutoFix-E uses a combination of enforced program invariants and software testing to craft sound repairs for bugs. The use of generic invariants to guide the patch generation process makes AutoFix-E generally applicable to the bug types found in Table 4.2, much like AE. Finally, Debroy and Wong use fault localization and program mutation to suggest patches in many of the same ways that AE does, suggesting it also fixes a similarly wide range of bugs. That is, AE is as general as the state of the art in theory, and is also as general as GenProg in practice in our evaluation. This anecdotal evidence suggests that AE is at least as general as the most widely-focused existing program repair techniques.

4.6.6 Qualitative Evaluation

Since GenProg has a strict superset of AE's mutation operators, any repair AE finds that GenProg does not is attributable to Fault, Fix, RepairStrat or TestStrat. We examine one such case in detail, related to command-line argument orderings and standard input in **`gzip`**.[3]

An exhaustive evaluation of all first-order mutations finds that only 46 out of GenProg's 75905 candidates are valid repairs (0.06%). Worse, the weightings from GenProg's repair strategy heuristics (which tier edit types after fault localization) are ineffective in this case, resulting in a 0.03% chance of selecting such an edit. GenProg considered over 650 mutants per hour, but failed to find a repair in time. By contrast, AE reduced the search space to 17655 via program equivalence and was able to evaluate over 5500 mutants per hour by careful test selection. However, AE's repair strategy was also relatively poor, considering 85% of possible candidates before finding a valid one. These results support our claims that while program repair could benefit from substantial improvements in fault localization and repair enumeration, our program equivalence and test strategies are effective in this domain.

[3] `http://lists.gnu.org/archive/html/bug-gzip/2008-10/msg00000.html`

4.7 Duality with Mutation Testing

At a high level, mutation testing creates a number of *mutants* of the input program and measures the fraction that fail (are *killed* by) at least one test case (the *mutation adequacy score*). Ideally, a high score indicates a high-quality test suite, and a high-quality test suite gives confidence about program correctness. By contrast, low scores can provide guidance to iteratively improve programs and test suites. Mutation testing is a large field, and characterizing all of the variations and possible uses of mutation testing is beyond the scope of this work; we necessarily adopt a broad view of the field and do not claim that our generalization applies in all cases. The interested reader is referred to Jia and Harman [56], to whom our presentation here is indebted, for a thorough treatment.

Broadly, we identify the mutants in mutation testing with the candidates in program repair. This leads to duality between the mutant-testing relationship (ideally all mutants fail at least one test) and the repair-testing relationship (ideally at least one candidate passes all tests).

4.7.1 Hypotheses

The competent programmer hypothesis (CPH) [83] and the coupling effect hypothesis [185] from mutation testing are both relevant to program repair. The CPH states that programmers are competent, and although they may have delivered a program with known or unknown faults, all faults can be corrected by syntactic changes, and thus mutation testing need only consider mutants made from such changes [56, p. 3]. The program is assumed to have no known faults with respect to the tests under consideration ("before starting the mutation analysis, this test set needs to be successfully executed against the original program . . . if p is incorrect, it has to be fixed before running other mutants" [56, p.5]). In contrast, program repair methods such as GenProg and AE assume that the program is buggy and fails at least one test on entry. GenProg and AE also assume the CPH. However, they often use operators that make tree-structured changes [129] (e.g., moving, deleting or rearranging large segments of code) or otherwise simulate how humans repair mistakes [58] (e.g., adding bounds checks) without introducing new ones. Search-based program repair further limits the set of mutants (candidate repairs) considered by using *fault localization*, program analysis that uses information from successful and failing tests to pinpoint likely defect locations, e.g., [37]), while mutation testing can consider all visited and reachable parts of the program (although profitable areas are certainly prioritized).

The coupling effect hypothesis (CEH) states that "complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults" [185]. Thus, even if mutation testing assesses a test suite to be of high quality using only simple mutants, one can have confidence that the test suite will also be of high quality with respect to complex (higher-order) mutants. For example, tests developed to kill simple mutants were also able to kill over 99% of second- and third-order mutants historically [185]. Following Offutt, we propose the following dual formulation, the search-based *program repair coupling effect hypothesis*: "complex

$$MT(P, \mathsf{Test}) = \mathsf{true} \text{ iff}$$
$$\left(\begin{array}{l} (\forall t \in \mathsf{Test}.\ t(P)) \ \wedge \\ (\forall m \in \mathsf{MTMut}(P).\ \exists t \in \mathsf{Test}.\ \neg t(m)) \end{array} \right) \implies \forall t \in \mathsf{FutureTest}.\ t(P)$$

$$PR(P, \mathsf{Test}, \mathsf{NTest}) = m \text{ iff}$$
$$\left(\begin{array}{l} (\forall t \in \mathsf{Test}.\ t(P)) \ \wedge \ (\forall t \in \mathsf{NTest}.\ \neg t(P)) \\ \ \wedge \ m \in \mathsf{PRMut}(P) \ \wedge \ (\forall t \in \mathsf{Test} \cup \mathsf{NTest}.\ t(m)) \end{array} \right) \implies \left(\begin{array}{l} (\forall t \in \mathsf{FutureTest}.\ t(P) \implies t(m)) \\ \ \wedge \ (\forall t \in \mathsf{FutureNTest}.\ t(m)) \end{array} \right)$$

Figure 4.4: Dual formulation of idealized mutation testing and idealized search-based program repair. Ideally, if mutation testing indicates that a test suite is of high quality ($MT(P, \mathsf{Test}) = \mathsf{true}$) then that suite should give us very high confidence of the program's correctness: passing that suite should imply passing all future scenarios. Dually (and ideally), if program repair succeeds at finding a repair ($PR(P, \mathsf{Test}) = m$) then that repair should address all present and future instances of that bug (pass all negative tests) while safeguarding all other behavior: if the original program would succeed at a test, so should the repair. The right-hand-side consequent clauses encode quality: a low-quality repair (perhaps resulting from inadequate Test) will appear to succeed but may degrade functionality or fail to repair the bug on unseen future scenarios, while low-quality mutation testing (perhaps resulting from inadequate MTMut) will appear to suggest that the test suite is of high quality when it fact it does not predict future success.

faults are coupled to simple faults in such a way that a set of mutation operators that can repair all simple faults in a program will be able to repair a high percentage of the complex faults." This formulation addresses some observations about earlier program repair results (e.g., "why is GenProg typically able to produce simple patches for bugs when humans used complex patches?" [47]).

Whether or not this program repair CEH is a true claim about real-world software system is unknown. While some evidence provides support (e.g., in Section 4.6, many faults repaired with higher-order edits can also be repaired with first-order edits), there is a troubling absence of evidence regarding repairs to complex faults. Broadly, current generate-and-validate program repair techniques can address about one-sixth to one-half of general defects [47, 58, 138]. It is unknown whether the rest require more time (cf. [174]) or better mutation operators (cf. [58]) or something else entirely. Since fixing (or understanding why one cannot fix) these remaining bugs is a critical challenge for program repair, we hope that this explicit formulation will inspire repair research to consider this question with the same rigor that the mutation testing community has applied to probing the CEH [185, 186].

4.7.2 Formulation

We highlight the duality of generate-and-validate repair and mutation testing in Figure 4.4, which formalizes ideal forms of mutation testing and program repair. Both mutation testing and program repair are concerned with functional quality and generality (e.g., a test suite mistakenly deemed adequate may not detect future faults; a repair mistakenly deemed adequate may not generalize or may not safeguard required behavior) which we encode explicitly by including terms denoting future (held-out) tests or scenarios.

Given a program P, a current test suite Test, a set of non-equivalent mutants produced by mutation testing operators MTMut, and a held-out future workload or test suite FutureTest, we formulate mutation testing as follows: Under idealized mutation testing, a test suite is of high quality if $MT(P, \mathsf{Test}) = \mathsf{true}$ holds. That is, if P passes all tests in Test and every mutant fails at least one test. In practice, the equivalent mutant problem implies that MTMut will contain equivalent mutants preventing a perfect score.

Similarly, given a program P, a current positive test suite encoding required behavior Test, a current negative test suite encoding the bug NTest, a held-out future workload or test suite FutureTest, and held-out future instances of the same bug FutureNTest, we formulate search-based program repair. Idealized program repair succeeds on mutation m ($PR(P, \mathsf{Test}, \mathsf{NTest}) = m$) if all four hypotheses (every positive test initially passes, every negative test initially fails, the repair can be found in the set of possible constructive mutations (edits), and the repair passes all tests) imply that the repair is of high quality. A high quality repair retains functionality by passing the same future tests that the original would, and it defeats future instances of the same bug.

A key observation is that our confidence in mutant testing increases with the set non-redundant mutants considered (MTMut), but our confidence in the quality of a program repair gains increases with the set of non-redundant tests (Test).[4] We find that |MTMut| is much greater than |Test| in practice. For example, the number of first-order mutants in our experiments typically exceeds the number of tests by an order of magnitude, as shown in Table 4.1. Thus, program repair has a relative advantage in terms of search: not all of PRMut need be considered as long as a repair is found that passes the test suite. Similarly, the dual of the basic mutation testing optimization that "a mutant need not be further tested after it has been killed by one test" is that "a candidate repair need not be further tested after it has been killed by one test." These asymmetrical search conditions (the enumeration of tests can stop as soon as one fails, and the enumeration of candidate repairs can stop as soon as one succeeds) form the heart of our adaptive search algorithm (see Section 4.5.1).

4.7.3 Implications

The formalism points to an asymmetry between the two paradigms, which we exploit in AE, namely, that the enumeration of tests can stop as soon as one fails (the mutation testing insight), and the enumeration of candidate repairs can stop as soon as one succeeds (the program repair insight). From this perspective, several optimizations in generate-and-validate repair can be seen as duals of existing optimizations in mutation testing, and additional techniques from mutation testing may suggest new avenues for continued improvements to program repair. We list five examples of the former and discuss the latter in Section 4.8:

[4] Our presentation follows the common practice of treating the test suite as an input but treating the mutation operators as part of the algorithm; this need not be the case, and mutation testing is often parametric with respect to the mutation operators used [49].

1. GenProg's use of three statement-level tree operators (mutations) to form PRMut is a dual of "selective mutation," in which a small set of operators is shown to generate MTMut without losing test effectiveness [87].

2. GenProg experiments that evaluate only a subset of PRMut with crossover disabled [174] are a dual of "mutant sampling," in which only a subset of MTMut is evaluated [187].

3. GenProg's use of multiple operations per mutant, gathered up over multiple generations, is essentially "higher-order mutation" [188]. Just as a subsuming higher-order mutation may be harder to kill than its component first-order mutations, so too may a higher-order repair be of higher quality than the individual first-order mutations from which it was constructed [56, p. 7].

4. Attempts to improve the objective (fitness) functions for program repair by considering sets of predicates over program variables instead of using all raw test cases [176] are a dual of "weak mutation" [189], in which a program is broken down into components, and mutants are only checked immediately after the execution point of the mutated component [56, p. 8].

5. AE's compilation of multiple candidate patches into a single program with run-time guards (see Section 4.5.1) is a direct adaptation of "super-mutant" or "schemata" techniques, by researchers such as Untch or Mathur, for compiling all possible mutants into a single program (e.g., [184]).

Finally, our use of approximate program equivalence is directly related to the "equivalent mutant problem" [56, p. 9], where mutation-testing regimes determine if a mutant is semantically equivalent to the original. AE's use of dataflow analysis techniques to approximate program equivalence for detecting equivalent repairs is thus exactly the dual of Baldwin and Sayward's use of such heuristics for detecting equivalent mutants [88]. Offutt and Craft evaluated six compiler optimizations that can be used to detect equivalent mutants (dead code, constant propagation, invariant propagation, common subexpression, loop invariant, hosting and sinking) and found that such compiler techniques could detect about half [89]. The domains are sufficiently different that their results do not apply directly: For example, Offutt and Craft find that only about 6% of mutants can be found equivalent via dead code analysis, whereas we find that significantly more candidate repairs can be found equivalent via dead code analysis. Similarly, our primary analysis (instruction scheduling), which works very well for program repair, is not among those considered by early work in mutation testing. In mutation testing, the equivalent mutant problem can be thought of as related to result quality, while in program repair, the dual issue is one of performance optimization by search space reduction.

4.8 Future Work

We leave as orthogonal work the critical subject of repair quality, but note that our algorithm with $k = 1$ produces repairs that are equivalent to minimized GenProg repairs. Similar program repairs have been successfully evaluated by Red Teams [39], held out test cases and fuzz testing [181], and human judgments of maintainability [43] and acceptability [58]. Even incorrect candidate patches cause bugs to be addressed more rapidly [57], so reducing the cost of repairs while maintaining quality is worthwhile.

While the work presented here reduces repair testing costs by an order of magnitude and monetary costs by a factor of three using only first-order repairs, there are many avenues for improvement. "Mutant clustering" selects subsets of mutants using clustering algorithms [49] (such that mutants in the same cluster are killed by similar sets of tests): such a technique could be adopted for our repair strategy (cluster candidate repairs by testing behavior and prioritize repairs that differ from previously investigated clusters). "Selective mutation" finds a small set of mutation operators that generate all possible mutants, often by mathematical models and statistical formulations [190]. Such techniques are appealing compared to post hoc measurements of operator effectiveness [174] and suggest a path to principled, weighted combinations of simple mutations [129] and complex templates [58], both of which are effective independently. Finally, "higher-order mutation" finds rarer higher order mutants corresponding to subtle faults and finds that higher-order mutants may be harder to kill than their first-order component [188]. This is similar to the issue in repair where two edits may be rquired to fix a bug, but each reduces quality individually (e.g., consider adding a `lock` and `unlock` to a critical section, where adding either one without the other deadlocks the program. Insights such as these may lead to significant improvements for current program repair methods which succeed on about 50% of attempted repairs [47, 137].

More generally, better equivalence approximations from mutation testing [90, 92, 93, 94] could be used to augment our instruction scheduling heuristic. Just as the CPH encourages mutation testing to favor local operations corresponding to simple bugs [83], program repair may benefit from higher-level structural mutations (e.g., introducing new types, changing function signatures, etc.), which are integral to many human repairs.

4.9 Conclusion

This chapter formalizes the important costs of generate-and-validate program repair, highlighting the dependencies among five elements: fault localization, possible repairs, the test suite, the repair selection strategy, and the test selection strategy. We introduced a deterministic repair algorithm based on those insights that can dynamically select tests and candidates based on the current history of the run. The algorithm computes the quotient space of candidate repairs with respect to an approximate program equivalence relation, using syntactic and dataflow analyses to avoid superfluous test

when the outcomes are provably already known. We evaluated the algorithm on 105 bugs in 5 million lines of code, comparing to GenProg. We find that our algorithm reduces the search space by an order of magnitude. Using only first-order edits, our algorithm finds most of the repairs found by GenProg, and it finds more repairs when GenProg is limited to first-order edits. The algorithm achieves these results by reducing the number of test suite evaluations required to find a repair by an order of magnitude and the monetary cost by a factor of three. Finally, we characterize generate-and-validate program repair as a dual of mutation testing, helping to explain current and past successes as well as opening the door to future advances.

We show that AE's expressive power allows it to theoretically fix as many bugs as GenProg and competing state-of-the-art program repair techniques, arguing for its generality. AE also works "off-the-shelf," requiring only commonly available software artifacts to generate patches, which strengthens its usability. Automated program repair techniques like AE attempt to ease the human burden associated with the maintenance process. However, automatically applying program changes like those described in this chapter may reduce the human intuition associated with software artifacts. This reduction of human intervention in the software maintenance process may have a negative effect on system quality as it relates to developer understanding and maintainability. The next chapter details a human study in which we measure the future maintainability (a human-centric concern) of automatically generated patches which aids in our comprehensive evaluation of the efficacy of such techniques in practice.

Chapter 5

A Human Study of Patch Maintainability

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

– Martin Fowler, 1999 [77]

5.1 Introduction

In Chapter 4 we describe an efficient method for generating repairs automatically. Earlier work also showed that patches can also be automatically generated using evolutionary techniques [47], dynamic program behavior modification [39], enforcement of explicit pre- and post-conditions [130], and program transformation guided by static analysis [26]. While these automatically generated patches may be functionally correct, little effort has been taken to date to evaluate the understandability of the resulting code. Many developers express natural reluctance about incorporating machine generated patches into their code bases [47]. This chapter explores future maintainability concerns as they relate to different kinds of defect patches.

Regardless of the provenance of a given patch, the quality of patched code critically affects software maintainability. There are a number of factors that contribute to patch quality. Perhaps most direct is a patch's impact on functional correctness: a high quality patch should bring an implementation more in line with its specification and requirements. Informally, a patch should fix the bug while retaining all other required functionality. However, even functionally correct patches can vary considerably in quality. Consider two patches that fix the same defect. One patch touches many lines, introduces several gotos and otherwise unstructured control flow, and contains no comments; the other is short and succinctly commented. Even if both have the same practical effect on program semantics, the first produces

code that is likely more difficult to read and understand in the future as compared to the second. In this chapter, we focus on patch quality as it relates to code understandability and, more broadly, software maintainability.

Software maintainability is a broad concept; it has been measured from many angles, using many different metrics [145, 147]. Despite ample effort in the area of code quality metrics, it has been noted that there is no adequate *a priori* descriptive metric for maintainability [148, 149, 150]. Sillito *et al.* study and categorize the questions developers actually ask about a code base while performing maintenance tasks [59]. For example, when looking to modify a fragment of code, programmers often ask "What is the control flow that leads to this program point?," or "What variables are in scope at this point?" We define maintainability by the ease and accuracy with which these formalized maintenance questions can be answered about a given piece of code. If developers answer such questions less accurately or less rapidly, we say the associated maintainability has decreased. We can ground such a metric by taking advantage of a "natural experiment": many human-written patches are later *reverted* and undone (e.g., [65]), representing an explicit loss of maintenance effort. By examining both a reverted patch and a patch that stood the test of time — for the same bug — we perform a controlled experiment and obtain a lens through which to study patch quality.

We use this framework to quantify patch quality as it relates to maintainability, and to study the relative quality of automatically generated patches as compared to human patches. To date, we are unaware of any human studies on the maintainability of patches in particular, or any studies that examine the differences between automatically generated patches and those created by humans. We hypothesize that participants will neither perceive nor expose a difference in maintainability between human-created and machine generated patches. We additionally propose to augment machine generated patches with synthesized, human-readable documentation that describes the effect and context of the change. We further hypothesize that adding this supplemental documentation to machine generated patches increases maintainability on average.

To test these hypotheses, we conduct a human study in which participants are presented with code and asked questions using Sillito *et al.*'s forms. We measure both accuracy and effort as proxies for maintainability; more maintainable code should admit more correct answers in less time. We show that human patches that were later reverted are generally less maintainable, according to these proxies, than those that were not reverted to establish that the metrics are well-grounded. To support the stated hypotheses, we measure the net changes in both accuracy and effort between the original faulty code and both human-created and machine generated patches, holding all other factors constant. After establishing the relative maintainability effect of different types of patches, we examine which characteristics of the code relate to maintainability directly, and compare them with participants' opinions as to what they thought affected patch quality.

We find that documentation-augmented machine generated patches are of equal or greater maintainability than human-created patches. This work thus supports the long- term viability and cost-effectiveness of automatic defect repair. Additionally, we identify several code features that correlate with maintainability, which can support better patch

generation — both manual and automatic — in the future. The main contributions of this chapter are:

- A novel technique for augmenting machine generated patches with automatically synthesized documentation. We focus on documenting both the context and the effect of a change, with the goal of increased maintainability.
- A human study of patch quality in which over 150 participants are shown patches to historical defects from large programs and asked questions indicative of those posed during real-world maintenance tasks.
- Statistically significant evidence that machine generated patches, when augmented with synthesized documentation, produce code that can be maintained with equal accuracy and less effort than the code produced by human-written patches. These results provide preliminary evidence that automatically generated patches may viably reduce long-term maintenance costs.
- A quantitative explanation of differences in patch maintainability in terms of measured code features. We contrast features that are predictive of actual human performance with features participants report as relevant.

5.2 Motivating Example

This section uses real-world bug fixes as examples to show that the effects of code patches on maintainability merit further study.

There are typically an infinite number of implementations adhering to any consistent specification. As such, there are typically a corresponding infinite number of functionally-correct patches for a given defect. For example, different patches may use different algorithms or data structures, reorder statements, include or remove dead code, or feature different commenting or indenting. Functionally equivalent patches may therefore have different effects on the code's readability or maintainability.

We present two distinct patches for a bug in the `php` scripting language interpreter to illustrate this point concretely. The `substr_compare` function compares two string parameters, `main_str` and `str`, for equality. The length of the comparison is controlled by a variable `len` — strings are trimmed to `len` characters before being compared. Bug report #54454 describes a defect in `substr_compare` where "if `main_str` is shorter than `str`, `substr_compare` [mistakenly] checks only up to the length of `main_str`." That is, if `main_str=''abc''` and `str=''abcde''`, `substr _compare` would erroneously return "true."[1] Informally, the bug is that `len` is set too low: checking only up to `len=3` does not reveal the differences between `''abc''` and `''abcde''`.

Figure 5.1 shows one candidate patch that changes the `substr _compare` function directly. Lines added as part of the patch are preceded by a + while removed lines are denoted by a `-`. This patch removes the conditional on lines 8–10, which allowed `len` to be set too low; any extra letters are thus accounted for, and the bug in question is fixed.

[1] https://bugs.php.net/bug.php?id=54454 as of Feb 2012

```
1     if (offset >= s1_len) {
2       php_error_docref(NULL TSRMLS_CC,
3           E_WARNING, "The start position
4           cannot exceed string length");
5       RETURN_FALSE;
6     }
7
8   - if (len > s1_len - offset) {
9   -  len = s1_len - offset;
10  - }
11
12    cmp_len = (uint) (len ? len :
13        MAX(s2_len, (s1_len - offset)));
```

Figure 5.1: Patch #1 for `php` bug #54454 and surrounding code context. The patch modifies the `substr_compare` function, removing lines 8, 9, and 10.

```
1   + len--;      /* Do set len = len - 1 */
2     if (mode & 2) {
3       for (i = len - 1; i >= 0; i--) {
4   -     if (mask[(unsigned char)c[i]]) {
5   -       len--;
6   -     } else
7           break;
8       }
9     }
```

Figure 5.2: Patch #2 for php bug #54454 and surrounding code context. The patch modifies function `php_trim`, removing lines 4, 5, and 6 while adding line 1.

By contrast, the patch in Figure 5.2 alters code in a local helper function related to string trimming. This patch causes `len` to always be decremented once before the loop, instead of once for every iteration of the loop in which a valid letter was found (since the strings are null-terminated, the single decrement is always allowed). `len` is thus, again, left sufficiently high to enable appropriate string comparison.

In terms of functional correctness, the two patches are equivalent: both produce code that passes the test case associated with the bug as well as the other 8,471 regression tests for the `php` interpreter. However, the resulting code may not be equally easy to reason about or maintain. What are the meaningful differences between the two patches, and how do these differences affect future maintenance tasks?

First, the patch contexts differ. The first patch applies directly to the 38-line `substr_compare` function; the second to a local helper function of comparable size (39 lines). In practice, this distinction means that the first patch changes `len` closer to its definition; the second changes it closer to its use. Both the size and granularity level of the enclosing code may contribute to maintainability. For instance, developers may find shorter functions easier to reason about and thus have no preference between the two patches based on function length. However, they may struggle with low-level, detail-oriented code and thus find the first patch more ultimately maintainable.

With respect to language constructs, the first patch strictly removes control flow and an assignment statement, while

the second patch moves a statement outside of two conditionals and a loop, altering rather than removing control flow. While removing control flow often makes reading and understanding code easier, the effect is not universal (e.g., a loop with a constant number of iterations may be easier to grasp than its unrolling, even though the unrolling has fewer tests and branches).

Additionally, the second patch includes a very simple comment describing the effect of the change, which is typically viewed as a benefit [153, 154], but it also leaves in dead code (the loop on lines 3–9 now has no effect), which may confuse later maintainers.

Finally, the patches have different origins. The patch in Figure 5.1 was created by a developer and has remained untouched since April 3rd, 2011 (we thus deem it "accepted"). The patch in Figure 5.2 was evolved by the GenProg tool [47, 181] and augmented with machine generated documentation.

These two patches were both subjects in our human study of patch quality, and clearly differ in several potentially important ways. However, it is not immediately clear how these differences affect maintainability. Perhaps surprisingly, in our study, participants were 0.25% *less* accurate when reasoning about the human-written patch (Figure 5.1) than about the original, while participants were 6.05% *more* accurate when reasoning about the machine generated patch with documentation (Figure 5.2) than about the original (questions asked were common to both patches).

This example demonstrates that multiple patches fixing the same defect can be functionally correct, but result in differently maintainable code. We desire a more formal description of the relationship between various features (e.g., comments, patch context, control flow, etc.) and maintainability. We thus detail our human study and the resulting data in Section 5.3 and Section 5.4, designed to directly measure one notion of patch quality.

5.3 Approach

In this section we describe our proposal to augment machine generated patches with synthesized documentation, as well as our human study to measure aspects of patch maintainability.

5.3.1 Synthesizing Documentation for Patches

Automated program repair approaches hold out the promise of reducing some software maintenance costs, freeing up developers to focus on more important bugs, or allowing developers to address more issues in the same amount of time, since adapting a candidate patch takes less effort than constructing one from nothing [57]. However, if machine generated patches are of poor quality and are harder to maintain than human-generated patches, their economic advantage disappears.

Automated repair techniques typically validate candidate patches against test suites (see Chapter 4), implicit specifications (e.g., [26]) or explicit specifications (e.g,. [130]). The quality of such patches with respect to *functional*

correctness only has been evaluated elsewhere and found to be human-competitive (e.g., against large held out test suites [181] or even against DARPA Red Teams [39]). Recall that human developers are not perfect. For example, a recent study of twelve years of patches to multiple free and commercial operating systems found that 15%–24% of human-written fixes for post-release bugs were "incorrect and have made impacts to end users." [65] As seen in Section 5.2, however, equally-correct patches may be more or less maintainable. In this chapter we do not further address the issue of functional correctness and instead restrict attention to aspects of maintainability. All patches we consider, whether human-written or machine generated, pass all available test cases.

We hypothesize that the maintainability of machine generated patches can be improved by augmenting them with synthesized, human-readable documentation explicating their effects and contexts. Based on Sillito *et al.*'s set of maintenance questions, we identify state and control flow as critical for many types of maintenance. We hypothesize that developers will find maintenance easier if they understand how a patch changes program state (e.g., alters the values of variables) or alters program control flow (e.g., the conditions under which statements may be executed) at run-time. To that end we desire human-readable documentation that summarizes what a patch does, as opposed to why it was made.

We adapt the DeltaDoc algorithm of Buse *et al.* [151] which synthesizes human-readable version control commit messages for object-oriented programs. Their algorithm is based on a combination of symbolic execution and code summarization. In essence, each symbolically executed statement is associated with its corresponding path predicate, and differences between the statements and predicates before and after applying a patch are summarized into human-readable documentation. See Section 2.5.2 for an outline of the DeltaDoc technique. Typical output is of the form "When calling `A()`, If `X`, do `Y` Instead of `Z`," where `X` is a path predicate and `Y` and `Z` are symbolic statement effects. We do not modify this basic output format, but instead widen the scope of program statements for which DeltaDoc generates documentation. The existing approach performs several optimizations with the goal of limiting the size of the output documentation. We make several changes to the technique as published to favor completeness over concision. The following changes were made:

- We alter the algorithm to report changes to all program statements, regardless of the length of the output to favor comprehensive understandability in lieu of brevity. Automatically generated patches are often short [129]; we claim it is more important in this context to capture and describe all details. In particular, we remove all Summarizing Transformations in DeltaDoc's *Statement Filters* category [151, Sec. 4.3.1]. As a result, documentation is generated for statements, such as assignments to local variables, that are neither method invocations nor field accesses nor `return` or `throw` statements.

- We do not use single-predicate transformations that result in loss of information due to duplication or suspected lack of relatedness to the changed statements. For example, we output "If `a=5` and `b=true`, return `a`" instead of "If

`a=5`, return `a`." Formally, this is a removal of DeltaDoc's third Summarizing Transformation in the *Single Predicate* category: "drop conditions that do not have at least one operand in documented statements" [151, Sec. 4.3.2].

- We do not "simplify" output by removing elements such as function call arguments. For example, we output "Always call `str_compare(main_str, str)`." instead of "Always call `str_compare()`." Formally, this corresponds to removing the *Simplification* category of DeltaDoc's Summarizing Transformations [151, Sec. 4.3.4].

- We avoid high-level simplification contingent on the length of the output, to favor a complete explanation over a concise one. The stated motivation of such simplification was that "this information often is sufficient to convey which components of the system were affected by the change when it would be impractical to describe the change precisely" [151, Sec. 4.3.4] — for the purposes of software maintenance we attempt to describe such changes precisely, even at the cost of verbosity.

We do not claim any novel results in the domain of documentation synthesis algorithms. Instead, we focus on the novel application of documentation synthesis to the problem of patch maintainability, and particularly to improve the quality of machine generated patches.

5.3.2 Human Study Protocol

Our goal is to measure the maintainability of patched code and understand why some types of patches may be more or less maintainable than others. We present human participants with segments of patched code and ask them maintenance questions about those segments. We measure participant accuracy and effort in answering those questions. The remainder of this subsection formalizes our human study protocol, the procedure for selecting and presenting patches, the formulation and selection of questions, and finally participant selection.

Maintainability is difficult to evaluate *a priori* [148,149,150]. In this dissertation, we avoid predicting maintainability based on indirect correlations with auxiliary code features (cf. [145]) and strive instead to measure it directly. We present a study to measure both objective and subjective notions related to patch quality. Our general approach is to measure human *effort* and *accuracy* when performing various maintenance-related tasks (i.e., answer questions such as those proposed by Sillito *et al.* [59]). We also collect subjective judgments such as participant evaluations of quality and confidence. Whenever possible, we control for accuracy (e.g., by giving participants unlimited time and/or restricting attention to equally-accurate answers). If participants are equally accurate when reasoning about Patch X and Patch Y (typically two functionally correct patches that both address the same defect), but reasoning about Patch X takes twice as long, then Patch X imposes a higher maintenance burden (i.e., is less easily maintainable).

In our human study protocol, participants were initially presented with a detailed list of instructions and a tutorial detailing the required format for all answers in addition to example questions and answers. This training helps ensure

Program	LOC	Tests	Defects	Human-Accepted Patches	Human-Reverted Patches	Machine-Generated Patches	Description
`gzip`	491,083	12	1	1	0	1	Compression utility
`libtiff`	77,258	78	7	7	0	7	Image processing utility
`lighttpd`	61,528	21	3	1	2	1	Webserver
`php`	1,046,421	8471	9	8	1	8	Language interpreter
`python`	407,917	355	1	1	0	1	Language interpreter
`wireshark`	2,812,340	63	11	0	11	0	Packet analyzer
total	4,896,547	9,000	32	18	14	18	

Table 5.1: A list of the subject programs we used as sources for patches in our human study, including the number of each type of patch used for each code base.

that delays or mistakes can be attributed to the patches and not to initial confusion or training effects (we address such threats explicitly in Section 5.5). Participants were instructed not to attempt to run the code or use any external resources during the study. The heart of the study consisted of sequentially presenting each participant with 23 partial C code files (sampled from among a total set of 114 files), each with a length of 50 lines. The number 23 was chosen based on initial timing estimates to keep the total task duration manageable. Each code segment had a corresponding code understanding question that focused the user on a single line of code. Participants were asked to complete three tasks for each code segment:

- Answer the code understanding question (in free form text)
- Give a subjective judgment of how confident they were in their answer (using a 1–5 Likert scale)
- Give a subjective judgment of how maintainable they felt the code in question was (using a 1–5 Likert scale). Note that "maintainability" was not defined for participants; they were forced to use their own intuitions.

We recorded both participants' accuracy when answering questions and the time it took them to reach an answer. As accuracy and effort represent the two major costs of the software maintenance cycle, together they can serve as measurable proxies for some aspects of "maintainability."

Finally, participants were presented with an exit survey containing questions about their computer science experience and personal opinions on the concept of maintainability.

5.3.3 Code Selection

To allow for a direct, controlled comparison between human-written patches and machine-generates patches, we used the benchmark suite presented by Le Goues *et al.* [47, Tab. 1]. The subject programs used thus come from several large open-source projects under ongoing development that include over 4 million lines of code and 9,000 test cases. Individual statistics for each program are presented in Table 5.1. We randomly selected 32 defects for which

both human-written and machine generated patches were available. Each defect had a priority/severity rating (where available) of at least three out of five, was sufficiently important for developers to fix manually, and was important enough to merit a checked-in test case. In addition, for each such defect we obtained the original code (i.e., the code for the first version just before the bug appeared) and, if possible, any human-written patches that had previously attempted to fix that bug but were reverted.

There are thus five distinct types of code collected and considered in this study:

- **Original** — defective, un-patched code used as a baseline for measuring relative changes in maintainability.
- **Human-Reverted** — human-created patches that were later reverted during the normal course of software maintenance.
- **Human-Accepted** — human patches that have not been reverted to date (at least six months).
- **Machine**[2] — minimized, machine generated patches produced by the GenProg tool, taken directly from the dataset of Le Goues *et al.* [47].
- **Machine+Doc** — machine generated patches as above, but augmented with synthesized, human-readable documentation describing the effect and context of the change (see Section 5.3.1).

For patches effecting multiple changes, we centered the 50-line context window around the change affecting the largest number of lines, breaking ties randomly. We explicitly include both undocumented and automatically documented machine generated patches to test the effects of synthesized documentation on automatically generated patches. However, we specifically do not test the effect of synthesized documentation on human generated patches, as the goal of this work is to compare fully automatic approaches with a completely manual one. For both Human-Reverted and Human-Accepted patches we add any relevant software versioning commit messages as comments so as to use all available human-created information associated with a given patch.

The types listed have natural overlap with respect to an individual bugs. For instance, a given defect might have corresponding code for the Original, Human-Accepted, Machine, and Machine+Doc categories. Participants were shown a randomly chosen sequence of code types. We ensured that no two code segments from a single bug would ever be presented to a single user to avoid any training bias for the code and bug in question.

Because the Machine patches were created using a C front end [100], they may not correspond exactly to the original code (e.g., they may have different indentation). We manually removed any non-original, unnecessary artifacts left by the tool. All patches presented to users were functionally identical to those produced by the GenProg tool, while

[2]Note that these experiments were performed prior to the development of AE (described in Chapter 4). However, all GenProg patches used for these experiments were single-edit repairs and thus equivalent to those AE might generate in practice. While this chapter studies patches specifically generated by GenProg, the results could be easily generalized to AE's patches based on expressive power.

syntactically matching the original code as closely as possible. No changes were made that could not have been applied mechanically.

As mentioned in Section 5.3.1, we hypothesize that the maintainability of machine generated patches will be improved by the addition of documentation summarizing effects and contexts. When presenting Machine+Doc code to participants, we inline the descriptive comments on (space permitting) or directly above the first line in the patch. Similarly, for human-written code, we inline the associated version control log message, if any. Thus, in terms of functional correctness and program semantics the Machine and Machine+Doc patches are identical: the only difference is the addition of documentation in the latter.

5.3.4 Code Understanding Question Selection and Formulation

To measure maintainability, we require subject questions that are indicative of those developers would ask during the maintenance process. Sillito *et al.* identify 44 different types of questions they directly observed real developers asking when performing maintenance tasks [59]. We used the five of these generic question types that focused on line-level granularity, as this was the most appropriate for 50-line code segments we presented to participants. Examples of the question types we selected are as follows:

- What conditions must hold to always reach line X during normal execution?
- What is the value of the variable "`y`" on line X?
- What conditions must be true for the function "`z()`" to always be called on line X?
- At line X, which variables must be in scope?
- Given the following values for relevant variables, what lines are executed by beginning at line X? `Y=5 && Z=true`.

Many of the questions observed by Sillito et al. were more general in nature and would not have been applicable for gauging humans' understanding of the code segments used in the study. An example of a question Sillito et al. observed that did not apply "Does this type have any siblings in the type hierarchy?" (question #9 from [59]). In the majority of cases, the code segments shown to participants did not represent an entire class and, as such, there generally would not have been enough context to answer such a question. Question types were randomly selected for each code segment collected. If a question type did not apply to the code in question (e.g. a question about function calls when none appeared in the code), a new question type was randomly selected until a viable option was found. We call the line X in the examples above the *focus line*.

As all questions operated at a line-level granularity, we applied a deterministic algorithm for choosing the focus line. Our goal when selecting focus lines is to direct participant attention to the changes made by the patch; directing them to

unchanged statements across different versions of similar code would fail to measure changes in maintainability caused by the patches. The main stipulation for choosing a line on which to focus was that it must occur in all relevant code segments (i.e., it must be associated with all available patches for that bug), to allow for controlled experimentation. For example, if the human-created and machine generated patch for a given bug share the same context, then only lines that occur in both patched versions of the code as well as the original source are valid choices. We adopted the following process for choosing the focus for a given patch's relevant code segments:

1. Let *S* be the intersection of all lines for all relevant code segments. If $S = \emptyset$, discard the bug and restart, otherwise proceed to step 2.

2. Let *T* be the subset lines in *S* that are *dominated* [191] by any part of the largest change from each of the relevant patches. If $T = \emptyset$, repeat with next largest change from the patch in conflict. If no further changes exist, discard the bug and restart, otherwise proceed to step 3.

3. Choose the line in *T* that is closest to a line changed by the patch (as reported by `diff`) in question.

Once both a focus line and a question have been selected for a given code segment, the remaining portions of the question were selected uniformly at random, but with the stipulation that the code changed for the patch be highlighted whenever possible. For instance, if the question was "what is the value of variable `y` on line X?," variable `y` was chosen randomly out of all the variables with values that were affected in the relevant patches. If no such variable existed or was in scope at line X, a variable was selected at random from those in scope. In the above example, the typical outcome of this process was that X referred to a line dominated by code changed by all of the patches (thus, if control flow were to reach line X, it must first have passed through patched code) and the variable `y` was one with a value altered by code changed by the patch.

Using this set of guidelines, we selected and crafted maintainability questions for a total of 114 code segments including 40 semantically distinct patches from the 32 unique bugs. When asked if they thought the study questions mimicked the actual maintenance process on a Likert scale (1–5), the majority of participants responded affirmatively.

5.3.5 Participant Selection

We aim to measure the maintainability of real source code. We thus require study participants with skills at least on par with novice developers, who might perform maintenance tasks on the target systems in the real world. We solicited responses from three groups of people and imposed accuracy standards to ensure that our results are more likely to generalize. All participants were required to have at least some programming experience in the C language. While participants were asked to self-report their experience, we used only objective accuracy measurements as cutoffs.

Participants fell into three categories: 27 fourth-year undergraduate students, 14 graduate students, and 116 Internet users participating via Amazon.com's Mechanical Turk "crowdsourcing" website. A fourth year computer science undergraduate student is indicative of someone who will soon enter the target industry as a novice. This is the lowest acceptable level of experience for our study and would represent "new hires" or developers who may not be familiar with the project — an especially important demographic to consider when studying the future maintainability of a system. Graduate students have generally had more experience and are more akin to a somewhat experienced developer from another project. Finally, Mechanical Turk participants varied widely in both industrial and academic experience. Participants were kept anonymous and were offered a chance for drawing in a $50 Amazon gift card and either class extra credit (via a randomized completion code, for students) or $4.00 (for Mechanical Turk participants).

In all cases, we impose an accuracy cutoff (described below) to ensure the quality and generalizability of the overall participant population. Amazon.com's Mechanical Turk crowdsourcing service deserves a detailed mention; it is effective as a means of obtaining a diverse population, but requires special consideration to ensure the overall quality of the data set. Previous work has shown that the Mechanical Turk participants can be effective when large populations are required [192, 193] — including for software engineering studies [41]. However, when offered a reward for an anonymous service, people may attempt to receive compensation without giving their best effort. We therefore set two criteria for *all* participants. First, participants were required to give answers for all questions and complete the exit survey fully. Second, participants who scored more than one standard deviation below the average student's score were removed from consideration. This accuracy cutoff was imposed because we cannot directly control for experience levels (especially with Mechanical Turk participants) and desire a level of competency consistent with someone who has completed at least some portion of an undergraduate education. Participants were aware of accuracy requirement and that their reward depended on it; since there was no time limit, participants were thus encouraged to take as long as necessary to get the correct answer, rather than to rush through the questions. In practice participants did just that (see details in Section 5.4), which often allowed us to measurably hold accuracy constant and thus use time taken as a key proxy for maintenance difficulty.

5.4 Experiments

This section presents statistical analyses of the results of the human study to address the following research questions:

1. How do different types of patches affect maintainability?

2. Which source code characteristics are predictive of our maintainability measurements?

3. Do participants' intuitions about maintainability and its causes agree with measured maintainability?

Recall that we are focusing only on particular aspects of maintainability (i.e., accuracy and effort when answering types of questions observed in real-world maintenance [59]).

As mentioned in Section 5.3.5, we impose an experimental cutoff to ensure only participants representative of the target "new developer" population are included in our analyses. Out of the 41 students surveyed, the mean accuracy was 53.8%. We thus establish a cutoff one standard deviation below that mean, at 34.7% accuracy. Imposing this cutoff restricted the overall subject pool from 157 to a total of 102 participants and over 2,100 individual data points.[3]

5.4.1 How do patch types affect maintainability?

We use two metrics to measure patch maintainability, corresponding to major costs throughout the maintenance process: accuracy and effort. *Accuracy* is calculated by manually verifying all collected responses and measuring the percentage of correct answers for all questions of a given patch type. At least two annotators verified each participant's answers to mitigate grading errors or ambiguities due to the use of free-form text. *Effort* is a calculated by averaging the number of minutes it took participants to submit a *correct* answer for all questions related to a given patch type. We omit incorrect answers as part of this statistic because incorrect answers often required only a few seconds if a participant decided to skip a question or simply guess.

Figure 5.3 shows the average percent change in accuracy for a given patch type. We measure the change in accuracy for a patch type t as follows:

$$\mathsf{change\_in\_accuracy}(t) = \frac{\sum_{i \in t} acc(patch_i) - acc(orig_i)}{|t|}$$

where i ranges over all patches of type t, $patch_i$ is a particular patch, and $orig_i$ is the corresponding original code. The acc function calculates the average accuracy for maintenance questions about a particular piece of code over all participants.

If all other variables are held equal (i.e., the code it modifies, any documentation it may include, etc.), any change in accuracy is explained only by the patch. A one-sided Student's t-test shows that none of the means of the percent changes presented can be considered different in a statistically significant manner ($p = 0.066 \not\leq 0.05$). Therefore, we cannot conclude that humans are more or less capable of correctly answering questions typical of those that arise during the maintenance process when examining different types of patches. However, we can conclude that accuracy on machine generated patches is *not worse* than accuracy on human-written patches (i.e., mean accuracy for machine generated patches is either greater than or equal to that for human-written patches).

[3]The data collected are available at http://www.cs.virginia.edu/~zpf5a/maintainability/data.html

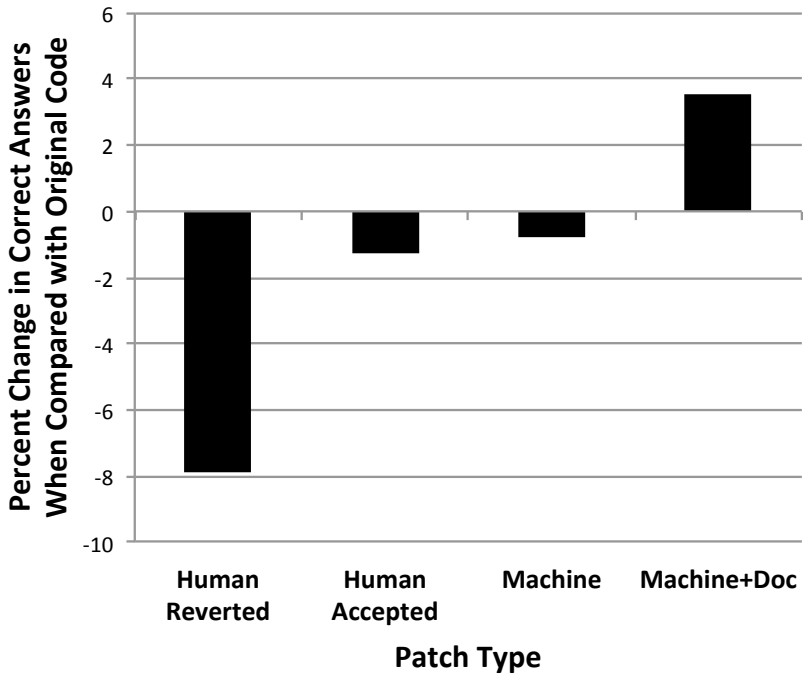

Figure 5.3: Percent change in *accuracy* of participants' answers as a function of the type of patch. The percent change is measured against the original, buggy code (i.e., before the patch was applied). Of the four types of patches we investigated, the sole type that, when applied, increases the maintainability of the original code on average is Machine+Doc. However, the means of the percent changes presented are not different in a statistically significant manner.

Figure 5.4 shows the percentage of time saved for each patch type when compared with the corresponding original code. The time saved is calculated as follows:

$$\text{time\_saved}(t) = \frac{\sum_{i \in t} time(orig_i) - time(patch_i)}{|t|}$$

where i ranges over patches of type t, $patch_i$ is a particular patch shown to participants, $orig_i$ is the corresponding original code, and the $time$ function returns the average time taken by all participants who answered the question correctly. In this measurement, the mean time required to correctly answer questions of Machine+Doc patches is lower than the mean time required to answer Human-Accepted patches in a statistically significant manner ($p < 0.048$). Specifically, Human-Accepted patches resulted in 20.9% increase in time-to-correct-answer, compared to the original code. However, Machine+Doc patches actually reduced the average time-to-correct-answer by 10.6%.

As mentioned in Section 5.3.3, we do not explicitly measure the effect of machine generated documentation on human patches. In Section 5.1, we hypothesized that we could create and present machine generated patches so that they would be at least as maintainable as comparable human-generated patches. As such, investigating the independent effect of synthesized documentation is outside of the scope of this work. We instead aim to provide evidence that fully

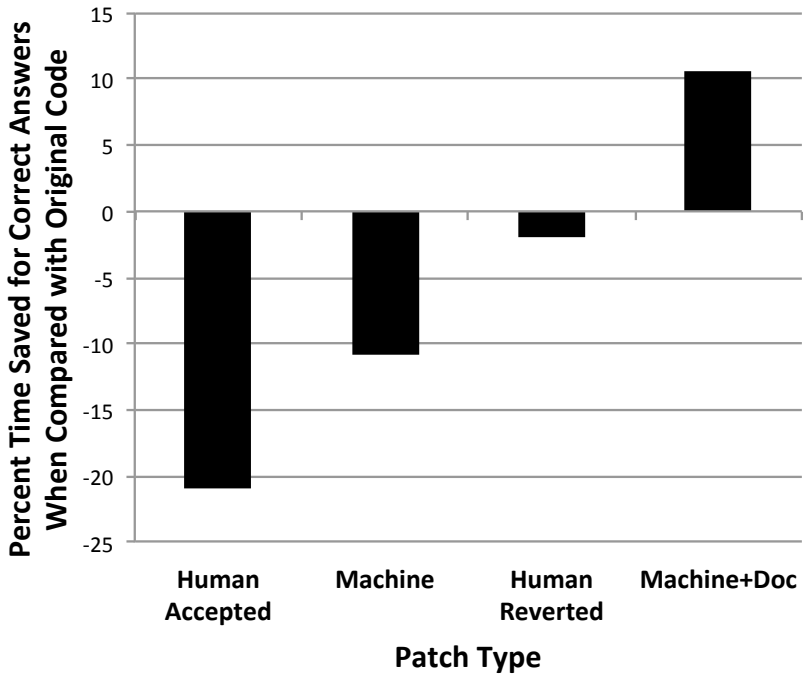

Figure 5.4: Percent *effort* saved for correct answers as a function of patch type. The percent change is measured against the original, buggy code (i.e., before the patch was applied). Of the four types of patches we investiaged, the only type that, on average, saved humans effort was Machine+Doc. With statistical significance, the mean effort saved when applying Machine+Doc patches is strictly greater than the effort saved when applying Human Accepted patches. In fact, we find that Human Accepted patches actually cause an increase in effort over the original code.

machine generated patches may be viable compared to human-generated patches.

To summarize, participants are *at least as accurate* when answering maintenance questions about machine generated patches augmented with documentation then when they are answering questions about human-written patches. In addition, when accuracy is held constant, participants take less time to correctly answer maintenance questions about machine generated patches augmented with documentation than they do to correctly answer questions about human-written patches (by 31.5%, $p < 0.048$). Perhaps surprisingly, for this aspect of maintainability (i.e., quickly and correctly answering questions that are indicative of the real-world maintenance process), Machine+Doc patches are more maintainable than Human-Accepted patches based on the results of this study: they admit at least as much accuracy and require less maintenance time on average with statistical significance.

5.4.2 Which code features predict maintainability?

We have established that patches from different sources have different maintainability, as measured by the accuracy obtained and effort required when participants answer indicative software maintenance questions about them. In this

section we investigate these changes in maintainability in terms of code features and quantitatively analyze those features in terms of their predictive power.

We tested several types of classifiers (e.g. naïve Bayesian, Bayesian network, multi-layer perceptron, decision tree, etc.) and chose a Logistic regression based on its simplicity and ability to correctly classify patches with respect to human accuracy (Bayesian and perceptron accuracies were within 9% of the chosen classifier). Given the feature values associated with a patch, the logistic model was able to correctly classify whether the participant would answer the maintenance question correctly for 73.16% of the 2109 human judgments. We find this model accurate enough to be useful when investigating the predictive power of individual features. We used the ReliefF technique to measure a given feature's relative predictive power, which computes this statistic without assuming conditional independence [194]. A principle component analysis shows that 17 code features are needed to account for 90% of the variance in the data. This high number of features highlights the complex nature of maintainability: it is not easily explained away by a small combination of notions.

Table 5.2 ranks the top 17 features by their ReliefF predictive power with respect to the human accuracy data collected from our study. The results show that a mix of both syntactical and semantic features help to predict maintainability. Of note is that "number of comments" is not a particularly predictive feature, while Figure 5.3 and Figure 5.4 show a clear difference in human performance between Machine and Machine+Doc — patch types in which the only difference is the presence of summarizing documentation. Heitlager *et al.* echo this intuition, finding that the majority of comments are "simply code that has been commented out" [147]. Since the mere *number* of comments present (i.e., one more in Machine+Doc) is not sufficient to explain the difference in human performance, we conclude that the *content* of the comments is critical.[4] That is, Machine and Machine+Doc can be viewed as a controlled experiment in which only the presence of the documenting comment changes, and thus only a few features change (e.g., number of comments and total number of characters). Those changed features are not sufficient to mathematically predict the differences in accuracy and effort actually observed; we thus conclude that an additional feature, such as the content of the comment, must matter. Together with Figure 5.3 and Figure 5.4, this result supports our claim that our proposal to augment machine generated patches with documentation that summarizes their effects and contexts is useful.

Previous investigations have indicated a lack of consensus on which metrics and concepts adequately explain maintainability [148, 149, 150]. The large number of low-impact features describing human accuracy, as measured by our study, reinforces this conclusion. In the following subsection, we elaborate this claim by showing that humans often fail to recognize which features actually predict maintainability.

[4] We cannot include comment quality as a feature since it does not admit a simple and standard numerical measurement.

Measured Code Feature	Power
Ratio of variable uses per assignment	0.178
Code readability (Buse *et al.* metric [195])	0.157
Ratio of variables declared out of scope / in scope	0.146
Number of total tokens	0.097
Number of non-whitespace characters	0.090
Number of macro uses	0.080
Average token length	0.078
Average line length	0.072
Number of conditionals	0.070
Number of variable declarations or assignments	0.056
Maximum conditional clauses on any path	0.055
Number of blank lines	0.054
Number of variable uses	0.041
Maximum statement nesting depth	0.033
Number of comments	0.022
Curly braces on same line as conditionals	0.014
Majority of lines are longer than half-screen width	0.012

Table 5.2: Relative Predictive power of code features for modeling human accuracy when answering questions related to maintainability (ReliefF method). A value of 1.0 indicates the feature is a perfect predictor while a value of 0.0 suggests the feature is of no value when predicting the given response variable.

5.4.3 Do human maintenance intuitions match reality?

Software maintenance is a complex and demanding task that may be poorly understood by some practitioners. We hypothesize that, in addition to not being able to identify what makes code more or less maintainable, humans may not be able to recognize maintainable code at all. This section compared subjective data collected during the study against measured human effort and accuracy to assess the validity of participants' intuitions about maintainability.

For each code segment in the study, participants were asked to provide not only an answer to the given question but also their confidence in their answer and a subjective judgment of how maintainable they believe the subject code to be (see Section 5.3.2). We found that participants' confidence in their answer correlated with their actual accuracy at a level of 0.18 ($p < 0.001$) using Pearson's product-moment statistic and with time to a correct answer at a level of -0.05 ($p < 0.05$). While there is no universal standard for the strength of a correlation, it is generally accepted that values between -0.3 and 0.3 are "not correlated" [196, 197]. We can thus conclude that participant confidence and participant accuracy are largely not linearly related.

Similarly, subjectively reported values for maintainability exhibited a Pearson correlation of 0.13 ($p < 0.001$) with actual accuracy and of -0.04 ($p < 0.20$) with time to correct answer. We conclude that participant judgments of the task difficulty and their own actual performance are also largely not linearly related.

During the exit survey of the study, participants were asked to list all relevant code features that they felt affected maintainability in any way. The frequencies with which code features were reported are shown in Table 5.3. Because

Human Reported Feature	Votes	Power
Descriptive variable names	35	0.000*
Clear whitespace and indentation	25	0.003*
Presence of comments	25	0.022
Shorter functions	8	0.000*
Presence of nested conditionals	8	0.033
Presence of compiler directives / macros	7	0.080
Presence of global variables	5	0.146
Use of goto statements	5	0.000*
Lack of conditional complexity	5	0.055
Uniform use and format of curly braces	5	0.014

Table 5.3: The top ten most-reported features by human participants when asked to list features they felt affected code maintainability. The second column lists the number of human participants who mentioned that feature; the third lists the relative predictive power of that feature when modeling actual human accuracy (ReliefF method; cf. Table 1). Features marked with an asterisk lack significant predictive power in the logistic regression model and are thus cases where humans misjudge the factors that affect maintainability.

they are self-reported as free-form text, the descriptions of the features are slightly less formal, but overall participants felt that descriptive variable names, clear whitespace and indentation, and presence of comments affect maintainability the most. By comparing the "Power" column in Table 5.3 to the top 17 actually predictive features (Table 5.2), it can be seen that there is limited overlap between the set of features humans believe affect maintainability and those that are good predictors of it. For example, the feature most commonly mentioned by humans, "descriptive variable names," was manually annotated on the code snippets used in the human study by two separate annotators but was still found to have no predictive power. Similarly, the use of clear whitespace and indentation had a very minimal predictive effect — far below any of the top 17 predictive features. Of the three features most often reported by participants, only one (the presence of comments) has a significant predictive power.

5.4.4 Qualitative Analysis

We now present two case studies to help illustrate the perhaps-surprising result that the features participants claim are most influential with respect to maintainability do not uniformly result in higher accuracy. The human-created (and later reverted) patch shown in Figure 5.5 exhibits many of the features participants report make code more maintainable. For instance, comments (the third highest feature) account for almost half of the lines in the patch. Additionally, there is only a single one-letter potentially nondescript variable and, given the juxtaposition of its type, `GtkWidget`, the use of the letter `w` would not seem overly ambiguous in this case. It is clearly indented, uses whitespace well, is a short function, lacks gotos, and does not feature complex conditionals. Despite displaying qualities reported to aid in maintainability, participants correctly answered questions associated with this code only 20.0% of the time — significantly below overall average of 57.2% for all questions in the study. The original, un-patched version of this code contains less

```
1  - void file_save_cmd_cb(GtkWidget *w,
2  -                       gpointer data) {
3  + void file_save_cmd_cb(GtkWidget *w _U_,
4  +                       gpointer data _U_) {
5     /* If the file's already been
6        saved, do nothing.  */
7     if (cfile.user_saved)
8       return;
9  +  /*Properly dissect innerContextToken for
10 +   Kerberos in GSSAPI. Now, all I have to
11 +   do is modularize the Kerberos dissector*/
12    /* Do a "Save As". */
13 -  file_save_as_cmd(w, data);
14 +  file_save_as_cmd(after_save_no_action,
15 +        NULL, FALSE);
16 }
```

Figure 5.5: Human-Reverted patch from **wireshark**. The patch modifies function **file_save_cmd_cb**, replacing lines 1–2 with lines 3–4 in addition to replacing line 13 with lines 14–15 and adding the comment on lines 9–11. Despite containing many features subjectively associated with high maintainability, participant accuracy on this snippet was 37% lower than average.

commenting and shows a slightly greater use of nondescript identifiers (i.e., **w** and **data** vs. **after_save_no_action**). Despite this, participants exhibit 6.1% greater accuracy, on average, for questions about the original code.

Figure 5.6 presents a code segment which exhibits relatively few of the features participants claim help to increase maintainability. Notably, the code lacks comments entirely and most of the variable names are terse. While humans subjectively report that these features should make it difficult to answer questions correctly, the average accuracy rate associated with the corresponding maintenance question was 75% — or 17.8% above the average and a 55% increase over the code depicted in Figure 5.5. While the code in Figure 5.6 does not match human-reported notions of maintainability, it does have higher-than-average for three out of the top fives features shown to actually predict maintainability in Table 5.2.

We conclude that there is a significant disconnect between human intuitions and reality regarding which code features affect maintainability. This discrepancy reinforces the need to investigate the root causes of maintainability with respect to guiding future development of both human-created and machine generated patches. More directly, as automatically generated patches become more commonplace, it is increasingly critical to know which features actually affect maintainability. Automated repair approaches can often produce multiple patches or target certain code features. Our results suggest, for example, that machine generated patches should pay more attention to using locally scoped variables and keeping the total size of the code low than to avoiding nested or complex conditionals.

```
1  static void snmp_users_update_cb(
2      void* p _U_, const char** err) {
3    snmp_ue_assoc_t* ue = p;
4    GString* es = g_string_new("");
5    *err = NULL;
6    if (! ue->user.userName.len)
7      g_string_append(es,"no userName,");
8    if (es->len) {
9      g_string_truncate(es,es->len-2);
10     *err = ep_strdup(es->str);
11   }
12   g_string_free(es,TRUE);
```

Figure 5.6: Original un-patched code snippet from **wireshark**. Despite having few features reportedly associated with maintainability, this code was particularly easy for participants to reason about (75% accuracy).

5.5 Threats to Validity

Although our experiments show that automatically generated patches augmented with synthesized documentation are at least as maintainable as human written patches, our results may not generalize.

First, the code segments we selected may not be indicative of industrial systems. We attempted to address this threat by including code from a variety of application domains, including web servers, language interpreters, graphics processing, and compression utilities. However, our results may not generalize to commercially developed systems, closed-source programs, or programs with complex graphical user interfaces, for example. Furthermore, there are several threats related to failing to control for factors such as inherent code complexity or readability when measuring maintainability levels for various patch types. We attempt to mitigate these threats by randomizing the selection of both the code segments and the target questions when assigning tasks to patch types.

The participants selected may not accurately reflect industrial developers. We address this bias by soliciting a combination of senior level undergraduates, graduate students, and external participants. Participant self-reported computer science experience ranged from 1–35 years indicating a diverse population. We further restrict the population by removing participants whose skills may not be comparable with that of paid developers by imposing an accuracy cutoff. A related concern is that participants had no *a priori* experience with the code under study. Thus, while our experiments reflect situations in which developers are tasked with examining unfamiliar code, our results cannot generalize to maintenance tasks involving code developers are familiar with. Finally, the questions posed may not be indicative of all maintenance tasks. However, this dissertation focuses only on measuring maintainability as it relates to the questions developers ask when performing maintenance tasks as described by Sillito *et al.* [59] directly.

Two common threats associated with human studies are *training* or *fatigue* effects. A training effect occurs when participants do poorly at the beginning of a study and increase in accuracy with familiarity. Conversely, a fatigue effect occurs when participants grow tired or apathetic and their performance declines. We explicitly measured for these

effects and found none: accuracy changed by only half a percentage point on average between the first half and the second half of the study.

Feature selection admits bias if the particular features measured are chosen based on either the code or the questions being asked. We mitigate this threat by choosing as features the union of those mentioned by participants and those used in previous studies exploring the maintenance process [41, 195].

5.6 Summary and Conclusion

We have presented a human study of patch maintainability. Our study is large (157 humans participated; the most-accurate 102 produced over 2,100 data points), uses high-priority defects from realistic programs (4.8 million lines of code and 9000 tests), is controlled (we compare human-written to machine generated patches for the same defects), and is grounded (we use human-reverted patches as a baseline indicative of wasted maintenance effort). The results shed light on the relative accuracy and effort required for participants to answer indicative maintenance questions on patched code. We also contrast the code-level features that humans think influence maintainability with those that are actually predictive of their performance. We acknowledge the research area of automated patch generation and include machine generated patches in our study, proposing to augment them with human-readable synthesized documentation describing their effects and contexts.

When we control for accuracy, we find that it took participants 30% less time to correctly answer maintenance questions about machine generated patches with synthesized documentation than to correctly answer questions about human-written patches, in a statistically significant manner. We find that our approach to automatically documenting machine patches is critical to this increase. This result is particularly compelling in light of the general perception that machine generated patches lower code maintainability. Finally, we investigate code features related to human accuracy and find a strong disparity between what humans think matters to maintainability (e.g., shorter functions, the presence of comments, descriptive variable names) and what is actually predictive (e.g., how often variables are modified, how many referenced variables are locally scoped, etc.).

Understanding the maintainability of patches is crucial to software engineering, especially as automated program repair becomes more common. We believe this work provides a first step toward directly measuring the maintainability of patches, both human-written and machine generated, as well as proposing particular approaches and treatments (i.e., synthesizing documentation, focusing on particular code features) that repair techniques, developers, and educators can consider for maintainability in the future. By presenting evidence that humans understand machine-documented, automatically generated patches as well as those created by humans we complement the empirical evidence that such patches are successful at fixing bugs in Chapter 4 to provide a comprehensive evaluation of automatic program repair.

Chapter 6

Conclusions

"A conclusion is the place where you got tired thinking."

– Martin H. Fischer

SOFTWARE maintenance is the dominant cost throughout the software lifecycle [2]. While previous work has helped to facilitate the maintenance process, many of the associated tasks are still performed manually by humans. The goal of this dissertation is to propose end-to-end automated improvements to the maintenance process to reduce both human-based and computational costs. A complimentary goal of this work concerns comprehensive evaluation of the described techniques, investigating both empirical cost savings and also human-based factors related to usability and quality of results. We summarize the contributions set forth in this dissertation in Section 6.1 and provide final remarks in Section 6.2.

6.1 Summary

This dissertation began by explaining both the costs associated with the software maintenance process and also the shortcomings inherent in many contemporary maintenance tools. We identified three opportunities for improvement with respect to established software maintenance tools:

1. **Generality.** Many tools focus on a particular context or constrained situation in the maintenance process to increase effectiveness while often decreasing wide applicability. We argue that this narrow scoping requires developers to use a large set of diverse tools to effectively automate the maintenance process and thus necessarily complicates their workflows while potentially adding hidden costs because of additional developer burden.

2. **Comprehensive Evaluation.** Traditional evaluations of maintenance tools are often limited to empirical evaluations only comparing against the current state-of-the-art in a specific problem domain. However, software

tools are ultimately designed to be used by and for humans. Thus, we believe that a comprehensive evaluation of such tools should address practical concerns like quality of results, understandability, and continued system maintainability.

3. **Usability.** Software maintenance tools often automate a certain task while introducing an additional human burden (e.g., an annotation or a feedback loop) as part of the overall technique. Such tradeoffs can hamper adoption by replacing one workflow task with another rather than strictly removing manual human effort.

We shaped our overarching thesis in response to these historical concerns — it is as follows:

> **Thesis:** it is possible to construct usable and general light-weight analyses using both latent and explicit information present in software artifacts to aid in the finding and fixing of bugs, thus reducing costs associated with software maintenance in concrete ways.

We believe the work in this dissertation supports this thesis statement by presenting and evaluating techniques that reduce the cost of software maintenance. Chapter 3 details an approach for clustering duplicate machine generated defect reports to reduce the human-centric costs associated with bug triage and fixing. Chapter 4 presents improvements to a state-of-the-art automatic program repair technique that yields concrete bug fixing cost reductions. Finally, Chapter 5 describes an in-depth human study that suggests that automatically generated patches can be as maintainable as those written by humans, which provides evidence supporting our patch generation technique's long term efficacy.

The automated techniques and evaluations in this dissertation were designed with the three previously mentioned overarching goals in mind. We address these goals directly in the following ways:

1. **Generality.** The defect clustering tool described in Chapter 3 works generically on structured output produced by several static analysis tools and on all associated bug types. Similarly, our generic automated patch generation technique (described in Chapter 4) fixes at least as many (and sometimes more) types of bugs as previous approaches, by construction (see Section 4.6.5).

2. **Comprehensive Evaluation.** In addition to traditional quantitative evaluations that focus largely on cost savings, we present evidence suggesting humans both agree with our techniques' results and also might use them in practice. Section 3.4.4 shows that humans overwhelmingly agree with our clustering technique while Chapter 5 provides evidence that our bug patches are as maintainable as those created by humans over time.

3. **Usability.** The techniques described in this dissertation function essentially "off-the-shelf" and do not specifically require any additional human input or intervention. By providing easily-usable techniques that strictly reduce the human burden associated with software maintenance, we hope to foster incremental adoption and thus increase the potential impact of such approaches.

Section 1.4 outlines several hypotheses about the software maintenance techniques described in this dissertation. We performed several empirical evaluations to test these hypotheses. A summarization of the contributions as they relate to each hypothesis is as follows:

- Hypothesis 1 — Our defect report clustering technique can cluster reports produced by many static analysis tools by construction; we have concretely demonstrated applicability on two such tools. Additionally, the technique can cluster defects of many types (including all those presented in Table 3.1), again by construction.

- Hypothesis 2 — In Section 4.6.5 we show that our automatic patch generation technique, AE, is at least as general (i.e., can fix at least as many types of bugs) as state-of-the-art approaches in practice.

- Hypothesis 3 — Section 3.4.2 presents evidence that our defect clustering technique is capable of clustering 60.6% of similar defects across a variety of Java and C programs with fewer than 5% false positives.

- Hypothesis 4 — Section 4.6.3 shows that AE, our automatic patch generation technique, reduces the cost of generating repairs when compared with the state-of-the-art GenProg technique by 70.2% (i.e., $4.40 vs. $14.78).

- Hypothesis 5 — We show that when presented with defect report clusters produced by our technique set to a low-false positive rate, developers agree with our clustering 99% of the time (see Section 3.4.4).

- Hypothesis 6 — Chapter 5 provides evidence to support the claim that our automatically generated patches, when augmented with basic machine generated documentation, are as maintainable (measured via human accuracy and effort for program understanding questions) as human-written patches in a statistically significant manner.

- Hypothesis 7 — We support the claim that our two techniques are usable throughout Chapter 3, Chapter 4, and Chapter 5. By construction, neither technique requires additional human input, which enhances their practicality. Also, by providing evidence in support of hypotheses 3–6, we gain confidence that the results are of adequate quality without specific tuning, both quantitatively and in terms of human-based notions of usefulness.

6.2 Discussion and final remarks

The cost of software maintenance not only dominates the overall cost of the software lifecycle [2] but also is a significant expense in practice. Bugs are being reported faster than they can be fixed [46, p. 363] and companies have gone so far as to pay outside developers up to $100,000 per bug fix to keep up with the high rate of incoming bug reports [67]. Software maintenance tools have helped to ease the human burden associated with finding and fixing bugs, however many of the associated tasks still require manual human effort. The work in this dissertation makes preliminary steps toward a more fully automated approach to software maintenance by tackling two largely-manual tasks: bug triage

and bug fixing. Our automatic defect report clustering technique can identify similar defects with few false positives to allow for parallelization of the bug triage and even repair process. Additionally, we present an automated program repair technique that patches roughly as many bugs as a comparable state-of-the-art technique but with 70% reduction in overall cost. We concretely show that these techniques produce results that humans deem to be both of high quality and also useful for facilitating software maintenance tasks. These combined quantitative and qualitative evaluations suggest that our automated techniques may be effective at reducing software maintenance costs in practice.

In addition to concrete cost reduction, we posed three high-level goals for the work in this dissertation: generality, comprehensive evaluation, and usability. We desire general techniques to simplify developers' workflows throughout the maintenance process. By introducing few new techniques that apply widely to many bugs, we impose minimal changes to existing processes while boosting potential cost savings. Comprehensively evaluating new maintenance techniques in terms of both cost savings and human-centric quality and usability is of paramount importance because of the wealth of human-based concerns associated with using such tools effectively. Software maintenance tools and techniques should also be highly usable, in terms of being easy to operate and understand as well as producing results that are applicable to the target maintenance concerns. Ensuring high usability can help to foster early adoption of such techniques and thus increase their potential impact, in practice. The previous section outlines ways in which we have emphasized these practical goals throughout this dissertation.

There are several peer-reviewed publications and technical reports that support the findings in this thesis — they are listed in Table 6.1. The steady increase in system size and complexity over time has translated into higher maintenance costs over time. Even in the short time over which the work presented in this dissertation was conducted, symptoms have arisen of ever-increasing maintenance concerns. For instance, software companies have increased the reward for bug bounties thirty-fold in just three years (i.e., 2010–2013). This suggests that leading practitioners recognize the growing cost associated with software maintenance and that research in this area will continue to have an impact in the industry. The work in this dissertation attempts to reduce the increasing costs associated with software maintenance in general and easy-to-use ways.

Venue	Publication
AOSD '07	*Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns* [35]
ICSM '10	*A Human Study of Fault Localization Accuracy* [41]
Tech report '12	*Fault Localization Using Textual Similarities* [42]
ISSTA '12	*A Human Study of Patch Maintainability* [43]
GPEM '13	*Software Mutational Robustness* [180]
WCRE '13	*Clustering Static Analysis Defect Reports to Reduce Maintenance Costs* [44]
ASE '13	*Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results* [131]

Table 6.1: This table presents the publications and technical reports supporting this dissertation.

Bibliography

[1] M. M. Lehman and F. N. Parr. Program evolution and its impact on software engineering. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 350–357, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[2] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[3] Jennie Baker. Experts battle £192bn loss to computer bugs. http://www.cambridge-news.co.uk/Education/Universities/Experts-battle-192bn-loss-to-computer-bugs-18122012.htm, 2013.

[4] Nathaniel Ayewah and William Pugh. The Google Findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, 2010.

[5] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, 2005.

[6] Atis telecom glossary. Technical report, 2011.

[7] Thomas Müller. Certified Tester Foundation Level Syllabus. Technical report, Board of International Software Testing Qualifications, 2011.

[8] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] L. A. Belady and M. M. Lehman. Programming system dynamics or the metadynamics of systems in maintenance and growth. Research report rc3546, IBM, 1971.

[11] C. V. Ramamoothy and W-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.

[12] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, pages 141–154, 2003.

[13] T.L. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[14] Symantec. Internet security threat report. In *http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf*, September 2006.

[15] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated Software Engineering*, pages 34–43, 2007.

[16] Patrick Francis and Laurie Williams. Determining "grim reaper" policies to prevent languishing bugs. In *International Conference on Software Maintenance*, pages 436–439. IEEE, 2013.

[17] Capers Jones. The economics of software maintenance in the twenty first century, 2006.

[18] BBC News. Microsoft Zune affected by 'bug'. In *http://news.bbc.co.uk/2/hi/technology/7806683.stm*, December 2008.

[19] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[20] Symantec. Internet security threat report, April 2011.

[21] Meir M. Lehman, Juan F. Ramil, P.D. Wernick, Dewayne E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. IEEE Symp. Software Metrics*, pages 20–32. IEEE Computer Society Press, 1997.

[22] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[23] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[24] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, April 2012.

[25] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: Explaining program failures via postmortem static analysis. In *Foundations of Software Engineering*, 2004.

[26] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

[27] Alexey Smirnov and Tzi-Cker Chiueh. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *Network and Distributed System Security Symposium*, 2005.

[28] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 3(6):41–49, 2005.

[29] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, 2006.

[30] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *International Conference on Sofware Engineering*, 2013.

[31] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, pages 342–351, New York, NY, USA, 2005. ACM.

[32] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering*, pages 30–39, 2003.

[33] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Notices*, 38(1):97–105, 2003.

[34] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[35] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Aspect-oriented Software Development*, pages 212–224, 2007.

[36] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with Jolt. In *European Conference on Object Oriented Programming*, 2011.

[37] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[38] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Foundations of Software Engineering*, pages 83–93, 2004.

[39] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, 2009.

[40] Raymond P. L. Buse, Caitlin Sadowski, and Westley Weimer. Benefits and barriers of user evaluation in software engineering research. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 643–656, 2011.

[41] Zachary P. Fry and Wes Weimer. A human study of fault localization accuracy. In *International Conference on Software Maintenance*, pages 1–10, 2010.

[42] Zachary P. Fry and Westley Weimer. Fault Localization Using Textual Similarities. *ArXiv e-prints*, 2012.

[43] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

[44] Zachary P. Fry and Westley Weimer. Clustering static analysis defect reports to reduce maintenance costs. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 282–291, 2013.

[45] Raymond P. L. Buse and Thomas Zimmermann. Information needs for software development analytics. In *International Conference on Software Engineering*, pages 987–996, 2012.

[46] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[47] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, pages 3–13, 2012.

[48] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, 2001.

[49] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference*, pages 94–98, 2008.

[50] Claire Le Goues. *Automatic Program Repair Using Genetic Programming*. PhD thesis, University of Virginia, 2014.

[51] Lori Pollock, K. Vijay-Shanker, David Shepherd, Emily Hill, Zachary P. Fry, and Kishen Maloor. Introducing natural language program analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 15–16, 2007.

[52] Lori Pollock. Leveraging natural language analysis of software: Achievements, challenges, and opportunities. *2013 IEEE International Conference on Software Maintenance*, page 4, 2012.

[53] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.

[54] V. Basili H. Rombach. Quantitative assessment of maintenance: an industrial case study. In *Conference on Software Maintenance*, pages 134–144, 1987.

[55] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, 2006.

[56] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[57] Westley Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[58] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Sofware Engineering*, 2013.

[59] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Foundations of Software Engineering*, pages 23–34, 2006.

[60] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, pages 492–497, 1976.

[61] Sharron Ann Danis. Rear Admiral Grace Murray Hopper, 1997.

[62] Mozilla Bugzilla. Reporting and charting. In https://bugzilla.mozilla.org/report.cgi, March 2014.

[63] Apple issues fix to reported OS X security hole. In http://www.bbc.com/news/technology-26335701, February 2014.

[64] Lily Hay Newman. Here's what you should know about Apple's security weakness. In http://www.slate.com/blogs/future_tense/2014/02/24/apple_s_security_flaw_ssl_vulnerability_how_do_i_protect_myself.html, February 2014.

[65] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *Foundations of Software Engineering*, pages 26–36, 2011.

[66] Apache OpenOffice Bugzilla. Reporting and charting. In https://issues.apache.org/ooo/, November 2012. Data collected by examining each available versions of the code from 01-01-2000 until 11-27-2012, comparing bugs filed against each version, counting "new" (CONFIRMED and UNCONFIRMED) bugs and "resolved" (RESOLVED) bug, aggregating the counts over time to produce the graph.

[67] Microsoft. Microsoft bounty programs. In http://technet.microsoft.com/en-US/security/dn425036, June 2013.

[68] H. D. Benington. Production of large computer programs. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 299–310, 1987.

[69] William H. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, 1998.

[70] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

[71] Raymond P.L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, November 2009.

[72] Raymond P.L. Buse. *Automatically Describing Program Structure and Behavior*. PhD thesis, University of Virginia, 2012.

[73] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, June 1974.

[75] eclipse.org. Eclipse platform technical overview. http://eclipse.org. Technical report, 2003.

[76] Architexa. Technical report, 2013.

[77] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[78] Mary Jean Harrold. Testing: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 61–72. ACM, 2000.

[79] Seifedine Kadry. A new proposed technique to improve software regression testing cost. *CoRR*, abs/1111.5640, 2011.

[80] Mary Lou Soffa, Aditya P. Mathur, and Neelam Gupta. Generating test data for branch coverage. In *Automated Software Engineering*, page 219, 2000.

[81] Amitabh Srivastava. Engineering quality software. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 11–11. Springer Berlin Heidelberg, 2004.

[82] Robert Geist, Jefferson A. Offutt, and Frederick C. Harris Jr. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computing*, 41(5):550–558, 1992.

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

[84] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18, 1982.

[85] Allen T. Acree. *On Mutation.* PhD thesis, Georgia Tech, 1980.

[86] Timothy A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, Connecticut, 1980.

[87] Aditya P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference*, pages 604–605, 1991.

[88] D. Baldwin and F.G. Sayward. *Heuristics for Determining Equivalence of Program Mutations*. Department of Computer Science: Research report. Yale University, 1979.

[89] A.J. Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[90] A.J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

[91] Rob Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[92] J.M. Voas and G. McGraw. *Software fault injection: inoculating programs against errors*. Wiley Computer Pub., 1998.

[93] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference*, pages 1338–1349, 2004.

[94] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.

[95] Francesco Logozzo and Manuel Fähndrich, editors. *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*. Springer, 2013.

[96] Mauro Pezzè and Mark Harman, editors. *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. ACM, 2013.

[97] Melinda-Carol Ballou. Improving software quality to drive business agility. White paper, International Data Corporation, June 2008.

[98] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[99] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[100] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, pages 213–228, 2002.

[101] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathemathical Society*, 74:358–366, 1953.

[102] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[103] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Principles of Programming Languages*, pages 174–186, 1997.

[104] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages*, pages 1–3, 2002.

[105] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[106] C. A. R. Hoare. An axiomatic basis for computer programming. *Communcations of the ACM*, 12(10):576–580, 1969.

[107] Goran Frehse, Colas Guernic, Alexandre Donz, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 379–395. Springer Berlin Heidelberg, 2011.

[108] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[109] Glenn Ammons, David Mandein, Rastislav Bodik, and James Larus. Debugging temporal specifications with concept analysis. In *Programming Language Design and Implementation*, San Diego, California, June 2003.

[110] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[111] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[112] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*, pages 500–517, 2001.

[113] Xiang Yin and John C. Knight. Formal verification of large software systems. In *NASA Formal Methods*, NASA Conference Proceedings, pages 192–201, 2010.

[114] Spec#. Technical report, 2014.

[115] Xiang Yin, John C. Knight, and Westley Weimer. Exploiting refactoring in formal verification. In *International Conference on Dependable Systems and Networks*, pages 53–62, 2009.

[116] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[117] Eric S. Raymond. The cathedral and the bazaar. In *Linux Kongress*, 1997.

[118] N. Bettenburg, R. Premraj, T. Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful...really? In *International Conference on Software Maintenance*, pages 337 –345, 2008.

[119] Nicholas Jalbert and Westley Weimer. Automated duplicate detection for bug tracking systems. In *International Conference on Dependable Systems and Networks*, pages 52–61, 2008.

[120] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *International Conference on Software Engineering*, pages 461–470, 2008.

[121] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *International Conference on Software Engineering*, pages 45–54. ACM, 2010.

[122] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Computer supported cooperative work*, pages 301–310, 2010.

[123] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *International Conference on Software Engineering*, pages 284–292, 2005.

[124] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[125] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. Free Software Foundation, 2002.

[126] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.

[127] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100, 2007.

[128] TD LaToza and Brad A Myers. Developers ask reachability questions. In *International Conference on Software Engineering*, pages 184–194, 2010.

[129] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[130] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[131] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering*, pages 356–366, 2013.

[132] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[133] Andrea Arcuri. On the automation of fixing software bugs. In *Doctoral Symposium — International Conference on Software Engineering*, 2008.

[134] David R. White, Andrea Arcuri, and John A. Clark. Evolutionary improvement of programs. *Transactions on Evolutionary Computation*, 15(4):515–538, 2011.

[135] Michael Orlov and Moshe Sipper. Flight of the FINCH through the Java wilderness. *Transactions on Evolutionary Computation*, 15(2):166–192, 2011.

[136] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *International Conference on Sofware Engineering*, 2013.

[137] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *International Conference on Sofware Engineering*, pages 772–781, 2013.

[138] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[139] Westley Weimer. Advances in automated program repair and a call to arms. In *International Symposium on Search Based Software Engineering*, pages 1–3, 2013.

[140] Stephen Cook, He Ji, and Rachel Harrison. Software evolution and software evolvability. Technical report, 2000.

[141] General Services Administration. Teleocmmunicatins: Glossary of telecommunication terms. Technical Report Federal Standard 1037C, National Communications System Technology & Standards Division, August 1996.

[142] Darrell R. Raymond. Reading source code. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 3–16, 1991.

[143] Spencer Rugaber. The use of domain knowledge in program understanding. *Ann. Softw. Eng.*, 9(1-4):143–192, 2000.

[144] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Reliability and Maintainability Symposium*, pages 235 –241, 2002.

[145] Kurt D. Welker, Paul W. Oman, and Gerald G. Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.

[146] M.H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[147] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *International Conference on Quality of Information and Communications Technology*, pages 30–39, 2007.

[148] Denis Kozlov, Jussi Koskinen, Markku Sakkinen, and Jouni Markkula. Assessing maintainability change over multiple software releases. *Journal of Software Maintenance and Evolution*, 20:31–58, January 2008.

[149] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[150] Kazuki Nishizono, Shuji Morisaki, Rodrigo Vivanco, and Kenichi Matsumoto. Source code comprehension strategies and metrics to predict comprehension effort in software maintenance and evolution tasks — an empirical study with industry practitioners. In *International Conference on Software Maintenance*, pages 473 –481, sept. 2011.

[151] Raymond P. L. Buse and Westley Weimer. Automatically documenting program changes. In *Automated Software Engineering*, pages 33–42, 2010.

[152] Victor H. Yngve and Jean E. Sammet. Toward better documentation of programming languages: Introduction. *Commun. ACM*, 6(3):76–, March 1963.

[153] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[154] David G. Novick and Karen Ward. What users say they want in documentation. In *International Conference on Design of Communication*, pages 84–91, 2006.

[155] NASA Software Reuse Working Group. Software reuse survey. In *http://www.esdswg.com/softwarereuse/Resources/library/working_group_documents/survey2005*, 2005.

[156] Raymond P. L. Buse and Westley Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis*, pages 273–282, 2008.

[157] Raymond P. L. Buse and Westley Weimer. Synthesizing API usage examples. In *International Conference on Software Engineering*, pages 782–792, 2012.

[158] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Doubleday, New York, NY, USA, 1984.

[159] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Program Analysis for Software Tools and Engineering*, pages 1–8, 2007.

[160] Antonio Vetro, Marco Torchiano, and Maurizio Morisio. Assessing the precision of findbugs by mining java projects developed at a university. In *Mining Software Repositories*, pages 110–113, 2010.

[161] ConQAT. ConQAT. https://www.conqat.org/, 2011.

[162] PMD. PMD. http://pmd.sourceforge.net/pmd-5.0.0/, 2012.

[163] Checkstyle. Checkstyle. http://checkstyle.sourceforge.net/, 2011.

[164] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, October 2007.

[165] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[166] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[167] Michael Sipser. *Introduction to the Theory of Computation*. Second edition. 1997.

[168] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Workshop on Mining Software Repositories*, May 2007.

[169] Igor Kononenko. Estimating attributes: Analysis and extensions of relief. In Francesco Bergadano and Luc De Raedt, editors, *European Conference on Machine Learning*, pages 171–182. Springer, 1994.

[170] Marko Robnik-Sikonja and Igor Kononenko. An adaptation of relief for attribute estimation in regression. In Douglas H. Fisher, editor, *Fourteenth International Conference on Machine Learning*, pages 296–304. Morgan Kaufmann, 1997.

[171] Justus J Randolph. Free-marginal multirater kappa (multirater κ free): an alternative to Fleiss' fixed-marginal multirater kappa. In *Joensuu Learning and Instruction Symposium*, 2005.

[172] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[173] Stephanie Forrest, Westley Weimer, ThanhVu Nguyen, and Claire Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[174] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.

[175] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Congress on Evolutionary Computation*, pages 162–168, 2008.

[176] Ethan Fast, Claire Le Goues, Stephanie Forrest, and Westley Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, pages 965–972, 2010.

[177] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing*, 9(1):3–12, 2005.

[178] Tian-Li Yu, David E. Goldberg, and Kumara Sastry. Optimal sampling and speed-up for genetic algorithms on the sampled onemax problem. In *Genetic and Evolutionary Computation Conference*, pages 1554–1565, 2003.

[179] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *International Conference on Genetic Algorithms*, pages 184–192, 1995.

[180] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[181] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[182] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Notices*, 39(10):432–448, 2004.

[183] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *International Symposium on Software Testing and Analysis*, 2013.

[184] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. In *International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

[185] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, January 1992.

[186] A. Jefferson Offutt. The coupling effect: fact or fiction. *SIGSOFT Softw. Eng. Notes*, 14(8):131–140, November 1989.

[187] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Softw. Test., Verif. Reliab.*, 4(1):9–31, 1994.

[188] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[189] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, 8(4):371–379, 1982.

[190] Akbar Siami Namin, James H. Andrews, and Duncan J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *International Conference on Software Engineering*, 2008.

[191] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[192] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Conference on Human Factors in Computing Systems*, pages 453–456, 2008.

[193] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *Empirical Methods in Natural Language Processing*, 2008.

[194] Marko Robnik-Šikonja and Igor Kononenko. Theoretical and empirical analysis of ReliefF and RReliefF. *Mach. Learn.*, 53:23–69, 2003.

[195] Raymond P. L. Buse and Westley Weimer. A metric for software readability. In *International Symposium on Software Testing and Analysis*, pages 121–130, 2008.

[196] Jacob Cohen. *Statistical power analysis for the behavioral sciences, 2nd edidtion*. Routledge Academic, 1988.

[197] Lawrence L. Giventer. *Statistical Analysis in Public Administration*. Jones and Bartlett Publishers, 2007.