Leveraging Light-Weight Analyses to Aid Software Maintenance

Ph.D. Dissertation Proposal
Zachary P. Fry
zpf5a@virginia.edu

Abstract

Software maintenance can account for up to 90% of a system's life cycle cost. As a result, many automated techniques have been developed to reduce the overall effort necessary to sustain software over time. While many of these tools work well under certain circumstances, we believe that they could be improved by taking advantage of large untapped sources of unstructured information that result from natural software development. As the size and complexity of systems increases, taking advantage of such previously-overlooked information is of paramount importance.

The proposed research will design lightweight analyses to extract latent information encoded by humans in software development artifacts and thereby reduce the costs of software maintenance. We will design analyses that apply throughout the maintenance process, focusing on three areas: (1) triaging large collections of exposed defects; (2) guiding the search for automatic defect repairs; and (3) ensuring the continued consistency of system documentation over time.

For each area of proposed research we aim to improve upon existing approaches to reduce the total cost of software maintenance while requiring minimal additional developer effort. We will evaluate the proposed work using these criteria on a diverse set of real world programs totaling millions of lines of code to concretely show the reduction of maintenance cost and improvement upon existing processes.

1 Introduction

Software maintenance is the dominant cost in the life cycle of modern systems [10, 56]. In response, many tools have been developed to reduce the overwhelming cost of finding and fixing bugs (e.g. [4, 15, 20, 23, 30, 31, 45, 47, 53, 71]) and to ensure long-term system understandability and evolvability (e.g. [11, 12, 13, 41, 55, 58, 59]). Despite these advancements, developers still struggle to handle the volume of maintenance tasks that arise in practice [31].

While tools designed to facilitate the maintenance process are effective under certain circumstances, many remain impractical in the face of real world constraints. For example, certain formal techniques may not scale well to large programs with complex and diverse maintenance tasks [4, 53]. By contrast, some techniques *require* wide-spread adoption and a large user base to be effective [45]. There is a conceptual gap between the state-of-the-art work in many of these areas and full practical adoption. We aim to close that gap by using light-weight analyses to develop generalizable techniques that can leverage unstructured developer-created information and ease the maintenance burden from beginning to end.

Our key technical insight lies in the exploitation of human-centric sources of information in software artifacts and the application of lightweight analyses to find generic solutions for known problems throughout the maintenance process. Previous work has shown that human factors (whether process- or information-based) can greatly affect the maintenance process [27, 26]; we plan to improve maintenance tasks by analyzing this latent human-created information. We also propose techniques that purposely require very little additional time or developer effort; we hypothesize these will be broadly applicable to a variety of tasks and systems.

Specifically, we aim to reduce maintenance costs at multiple stages in the software maintenance lifecycle. First, we will expedite the bug triage and verification process by clustering highly related, automatically-generated defect reports using structured document comparison. Next, we will incorporate domain-specific

knowledge into an existing automatic program repair framework to evolve quick and compelling bug fixes. Finally, to foster persistent and proactive system quality, we will use concept recognition techniques to identify incomplete and inconsistent code documentation and suggest high-level corrections.

For each research thrust we propose to evaluate our technique against state-of-the-art processes and approaches to get a concrete idea of how much effort can be saved or how much additional work could be done. Additionally, each technique should be broadly applicable — we propose to cluster defects produced by a variety of tools, to help fix diverse bugs from various domains, and to identify several types of low-quality comments in many languages. By better informing the maintenance process overall, we hope to reduce the maintenance burden and ensure higher quality systems over time.

2 Research Overview and Challenges

We propose to reduce the cost of software maintenance by designing lightweight analyses to extract and analyze latent information encoded by humans in software development artifacts. Many existing software maintenance techniques have proven effective but remain infeasible for applications of a certain size or complexity. We aim to alleviate three such bottlenecks via lightweight analyses and machine learning techniques.

One over-arching intuition we have is that code inherently contains actionable artifacts left by humans that many existing techniques fail to recognize or use. Programmers encode their domain knowledge in the code they write and we hypothesize that extracting and using such information can yield a useful source of additional knowledge when performing software maintenance tasks.

2.1 Clustering Duplicate Automatically-Generated Defect Reports

Software defects are notoriously hard to expose and locate. In-house testing and formal verification methods can both be quite expensive. Similarly, relying on end-users to find and report bugs is both costly and detrimental to consumer perception and product quality. As a result, there are many techniques designed to automatically find such defects to aid the software maintenance process [5, 6, 7, 30, 45, 48]. However, such tools can suffer from false positives, spurious warnings, and duplicate reports, all of which negate potential savings in maintenance effort [2, 67]. While previous work has examined the problem of false positives, the issue of identifying *duplicate* automatically-generated defect reports remains largely unexplored.

Previous work has shown that code clones are prevalent in practice and that they can lead to defects when changes are not made universally across all instances of copied and pasted code [8]. Static analysis-based defect detection techniques can produce thousands of reports for a single project — all of which have to be manually examined and triaged. Large classes of related reports are often produced by static analysis defect location techniques because of the patterned-based nature of their search methods and the prevalence of code clones in real systems. In an examination of the output of two popular static bug finders when run on 14 large open-source programs, we found that over 30% of defect reports (over 2,600 actual reports) could be clustered in an effort to save time by handling similar defect reports aggregately.

Explicitly targeting both real and spurious reports, we desire a method for accurately clustering related defect reports to facilitate the maintenance process. In practice, such clusters could be triaged and even fixed or discarded aggregately, thus reducing the overall developer effort necessary to process automatically-produced defect reports. We know of no existing work specifically tailored to this problem and find that existing code clone detection techniques are ineffective because the types of information inherent in an automatically-generated defect report are sufficiently different than those expected by such tools.

2.2 Improved Fitness Functions for Automatic Program Repair

The topic of automated program repair has gained popularity as a viable defect fixing strategy in recent years [35, 50, 70, 71]. Despite ample progress in the area, evolving quick and easy repairs for complex bugs in large systems remains difficult in practice because of large search spaces and the complicated nature of some bugs and their respective fixes.

One state-of-the-art program repair technique, `GenProg`, exploits observed biological and genetic principles to evolve bug fixes from existing code [71]. A key component of this algorithm is the *fitness function* (i.e., relative *correctness*) of any individual candidate fix (i.e., program variant) throughout the process. `GenProg`'s current fitness representation is based on the number of regression test cases a given program variant passes. By promoting program variants with desirable qualities (e.g. correct functionality), bug fixes are evolved over time. While previous work has examined the possibility of improving fitness functions to speed up the process and elicit more fixes in practice [1, 24, 36, 73], there are still many bugs that `GenProg` fails to fix in a practical amount of time given ample resources [42].

The existing model of variant fitness in the `GenProg` framework assumes that all test cases are equally representative of the desired program behavior. We investigated the validity of this assumption by examining the mutants created when fixing 10 bugs from a previous study [42]. Using 6,675 mutants created by GenProg as a part of the bug fixing process, we found significant variation in the correlations (using the Pearson product moment coefficient) between the outcome of any single test case and the mutants' measured similarity with an eventual fix. 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 current scheme of using all test cases with equal weights may not be the optimal method of measuring fitness. Additionally, the existing notion of variant fitness does not universally exhibit high *fitness-distance correlation* [38]. That is, for a given variant, a higher measured fitness value does not necessarily indicate code changes that are closer to a fix for the defect in question. Using data collected from 20 bugs fixed in previous experiments [42], we calculated the current fitness-distance correlation of the `GenProg` tool to be 0.145 (in previous work, values between -0.15 and 0.15 are commonly considered "uncorrelated" [38]). We thus aim to design a new fitness function that more closely correlates with this notion of how close a given mutant is to a valid fix without *a priori* knowledge of what such a fix looks like.

2.3 Ensuring Documentation Completeness and Consistency

Software documentation, specifically in the form of code commenting, is essential to program understanding [21, 62, 65, 72]. Consistent documentation changes should coincide with code changes (such as patches to fix bugs) to ensure the continued quality and understandability of a code base. However, previous work has shown that comments very rarely co-evolve with code in real-world systems [25]. Even when specific documentation requirements are implemented, developers fail to also maintain documentation at all levels of granularity [40]. In previous work, we have found that humans overwhelming identify high quality documentation as one of the most important aspect of code maintainability [26]. Siy *et al.* similarly found that a non-trivial portion of the issues that arise during industrial code inspection concern the quality of documentation [57]. Recently, there has been preliminary work examining both the completeness of comments and the consistency of comment updates with respect to future defects [32, 41, 55]. In all cases, low-quality and inconsistently updated comments correlated with higher defect density.

Figure 1 and Figure 2 illustrate the types of documentation we propose to identify. These examples of inconsistent and incomplete comments, respectively, are taken directly from the Mozilla code base. Humans happened to notice both of these low quality comments and change them to better reflect the code being documented, but previous work suggests many such comments go unnoticed in practice [55]. Additionally, these examples are from an interface definition and a solitary method call, respectively. In both cases the associated code offers little information to aid in understanding, which makes complete and consistent documentation even more important. Low quality documentation like that depicted can lead to a lack of understanding and thus cause developers to introduce bugs. For this reason, we desire a way to automatically detect such comments to ensure persistent system quality.

Many tools have been developed to aid in maintenance tasks relating to editing code but relatively few exist to aid in documentation. Such tools have generally focused on automatically generating documentation where none exists [12, 13, 58, 59]. Preliminary studies have focused on very specific classes of incomplete or inconsistent comments (specifically, only for Javadoc elements [55] and less than 30 comments each in large, open source programs [63, 64]), but there exists no general approach for identifying low-quality comments

```
/* - COUNT: The amount contains the cumulative number of times some event
 * has occurred since the application started up.  For instance, a
 * reporter reporting the number of page faults since startup should have
 * units UNITS_COUNT. */
 * - COUNT: The amount is an instantaneous count of things currently in
 * existence.  For instance, the number of tabs currently open would have
 * units COUNT. */
...
```

```
1    const int32_t UNITS_BYTES = 0;
2    const int32_t UNITS_COUNT = 1;
```

Figure 1: An example of an inconsistent comment taken from the `nsIMemoryReporter.idl` file in the Mozilla project. The old, inconsistent comment appears highlighted in red while the human-corrected, consistent comment appears in blue. The commit message associated with change was "Update the comment for UNITS_COUNT in nsIMemoryReporter." The two comments give conflicting explanations of the variable UNITS_COUNT.

of arbitrary types. While these techniques apply formal, strict models of completeness and consistency, they mostly overlook the idea of "concept similarity" which could be used compare the language in the source code with that of the comments. Unlike existing techniques, this approach does not require the presence language-specific commenting paradigms or the existence of specific natural language patterns and can be used generically on any code or comment. Previous work suggest that up to 34% of comments may be incomplete and up to 67% may be inconsistent with respect to their associated code in practice [55] and thus we aim to develop more precise models of documentation quality.

3 Proposed Research

We propose three main research thrusts related to improving parts of the software maintenance process:

1. To cluster related automatically-generated defect reports.
2. To develop a better fitness function for evolutionary program repair.
3. To create a generic model of comment completeness and consistency.

In the rest of this section we describe our approach to each problem in detail and in Section 4 we lay out our experimental design and evaluation for each research thrust.

3.1 Analysis and Characterization

Despite attempts to automate software maintenance tasks, the overall process remains costly in practice. Automatic bug finders can greatly reduce the number of post-release defects in software, but can also produce thousands of defects that have to be manually inspected and triaged. Automatic program repair can evolve fixes for some of these defects, but fails to fix a substantial percentage of real-world bugs in practice. In such a time-constrained environment, developers often focus on fixing more bugs or adding new features in favor of ensuring persistent documentation and overall continued code quality.

Many existing automated maintenance processes favor absolute correctness over overall cost and total gain in system quality. Human-centric information may be under-utilized because it is mistakenly viewed as

```
// Reset currentURI.
// Reset currentURI. This creates a new session history entry with a new
// doc identifier, so we need to explicitly save and restore the old doc
// identifier (corresponding to the SHEntry at activeIndex) below.
browser.webNavigation.setCurrentURI(this._getURIFromString("about:blank
    "));
```

Figure 2: An example of an incomplete comment taken from the `nsSessionStore.js` file in the Mozilla project. The old, incomplete comment appears highlighted in red above while the human-corrected, complete comment appears in blue below. The commit message associated with this change was "Bug 647028 - Followup: Add comment." The additional information in the updated comment explains the method call's effect on the program state.

unreliable or too system-specific. We propose to develop light-weight techniques for extracting such information and using them to inform the maintenance process. By inferring structured representations for various software artifacts and drawing inspiration from areas like natural language processing and machine learning, we plan to augment existing automatic maintenance techniques to make them more effective at ensuring long-term system quality. More specifically, by streamlining various parts of the process we hope to triage bugs faster and more effectively, fix more of the resulting defects, and ensure post-change documentation consistency, which we hypothesize will result in higher long-term system quality.

3.1.1 Research: Clustering Duplicate Automatically-Generated Defect Reports

While automatic bug finding techniques are effective at identifying pre-release defects, the pattern-based nature of many such tools leads to large classes of highly related defect reports. When faced with thousands of such reports, triaging and possibly even fixing related reports aggregately can save considerable developer effort. Existing duplicate detection techniques (targeting either code clones or similar manually-created defect reports) take either whole contiguous code segments [14, 18, 51] or unstructured natural language text [33, 60, 68] as input. By contrast, static analysis-based defect finders output fragmented code lines in addition to contextual and semantic information related to the defect in question. Thus, we hypothesize that by exploiting the special structure of such tools' output we can accurately cluster defect reports to save effort.

We propose to construct a model of defect report similarity that compares individual document sub-parts using various lightweight similarity metrics. In previous work we have successfully measured aggregate document similarity by comparing document sup-parts and experimentally learning the relative importance of each sub-comparison. This divided approach admits the use of specific types of similarity metrics given the type of information present and helps to promote generality — the technique can accept a wide array of defect report formats as input, handling missing sub-components gracefully. While existing code clone or duplicate defect report detection techniques generally only focus on syntactic information, our approach will also consider semantic information. We thus hope to identify not only obviously similar clusters of defect reports but also those containing syntactically-unique but semantically-related defect reports.

3.1.2 Research: Improved Fitness Functions for Automatic Program Repair

Accurately evaluating the relative correctness or fitness of a given program variant is crucial to evolving repairs through genetically-inspired processes. The state-of-the-art method for measuring fitness simply counts the number of regression tests a given program variant passes to quantify repair suitability. This notion of fitness does not, however, take into account how close a given variant is to fixing the bug in

question. We hypothesize that by weighting test cases according to *both* their ability to guard existing desired program behavior and also their relative importance when fixing previous bugs, we can better inform the evolutionary bug fixing process.

We propose to use a bug fix dataset published in previous work [42] to develop and evaluate our more effective fitness function. First, we propose to overturn that work's assumption that all test cases are created equal. We will measure the relative importance of various test cases using the measured "strength" of a given variant (i.e. its actual, *post hoc* fitness). Any test case that is passed by variants with high measured fitness but fails on less-strong variants could then be weighted highly in our proposed fitness function (i.e, because of its high discriminatory power). In addition to identifying important test cases, we also propose to identifying fit variants directly based on their characteristics. We propose several possible plans for accomplishing this goal:

1. We will measure how many of a variant's structural mutations are part of any eventual successful minimized patch. Put another way, this measures the number of code changes shared by both the variant and any known fix for the associated bug. Thus, we can use knowledge of bugs that we can fix in the present to try to learn better fitness functions for fixing bugs discovered in the future (or any bug `GenProg` currently fails to find a patch for).

2. We plan to explore the possibility that neutral mutations (those that do not change program variant's performance on test cases when the mutation is applied) may add functional diversity and thus be beneficial. We hypothesize that bugs with complex fixes may benefit from larger, more sweeping functional changes like those made possible by neutral mutations. Applying many neutral mutations by definition does not degrade the program behavior as outlined by the regression test suite, but could introduce unspecified behavioral changes that, when combined over time, may eventually steer a genetic search towards fixes for complex bugs.

3. If we find that there is an intrinsic difference between the strategies needed to fix the bugs presented in previous work and those currently eluding state-of-the-art techniques, we plan to cast known correct human-crafted fixes in the format used by `GenProg` and similarly use these fixes to measure the subset of shared mutations as described above in item (1). That is, we plan to use human-written fixes as a ground truth for what it means to be close to a repair.

4. A more speculative strategy would be to examine human fixes, specifically noting desirable traits with respect to code changes that lead to patches. It is possible, for instance, that moving contiguous statements is beneficial when trying to evolve patches and thus groups of mutations that are locally close should be rewarded. Conversely, it is conceivable that moving certain types of statements (e.g. variable assignments) does not often coincide with bug fixes in real world patches and should thus be discounted as part of a fitness function. While it is difficult to enumerate all possible features in this scenario, we hypothesize that rigorous examination of many real world patches may bring such trends to light. While similar notions have previously been considered locally when selecting mutations [43] they have not been considered globally for evaluating fitness.

In addition to learning weights for test cases, we hypothesize that run-time program invariants may be learned to further inform a more effective fitness function. Such invariants have been successfully used to inform fault localization [17, 46], expose incorrect interface usage [5], and guide the development of high-coverage test suites [3] in the past. For a given program, we can mine run-time invariants with values that correlate with the measured strength of established variants from bugs previously fixed in the system. Thus, any program variant that establishes an invariant known to be crucial when fixing similar bugs previously could be rewarded by our new fitness function.

Finally, the previously mentioned hypotheses assume that test cases and invariants found to be related to fixing a given bug will be similarly helpful in fixing similar future bugs in the same program. We will explicitly evaluate this hypothesis to ensure that these properties are generalizable across different bugs. We hypothesize that a combination of these insights will allow us to evolve patches for bugs that `GenProg` was

previously unable to fix while also speeding up existing fixes by more effectively measuring the fitness of program variants throughout the evolutionary process.

3.1.3 Research: Ensuring Documentation Consistency

Ensuring that documentation completely and consistently describes its associated code amidst a high volume of code changes is paramount to ensuring the long-term quality of a system [21, 32, 41, 55, 62]. We propose to develop a concept-based method for identifying incomplete and inconsistent comments, leveraging both automatic documentation techniques and lightweight similarity metrics. A concept-based method takes into account the frequency, specificity, and locality of the language used in documentation or source code to identify an arbitrary number of high-level, abstract *concepts* (comprised of both objects and actions) relevant to the text in question. This differs from existing approaches that attempt to describe a predetermined set of concepts based on code structure in that it can automatically *infer* important, abstract concepts that should be documented (rather than simply require documentation for all function arguments and return types, for instance). Established automatic documentation techniques necessarily identify concepts in code that are deemed relevant to its functionality. We will first use these tools to obtain an "oracle" set of important concepts for each function. We can thus compare existing comments with this comprehensive notion of what information an ideal comment would contain to determine if the existing documentation adequately and correctly describe the associated code.

Both existing comments and automatically generated documentation can be viewed as documents for which we will measure similarity. We hypothesize that a combination of established language similarity metrics (e.g. latent semantic analysis (LSA) [22], latent Dirichlet allocation (LDA) [9], TF-IDF [37], etc.) can adequately identify incomplete and inconsistent comments by comparing them with the ideal generated documentation. While a single metric may prove sufficient, the problem might require a weighted combination of multiple metrics [54]. By using an annotated set of high-quality, incomplete, and inconsistent comments we can experimentally determine a similarity cutoff for which our technique achieves the highest accuracy when attempting to identify low-quality comments.

In the event that we cannot accurately identify inconsistent comments by comparing oracle documentation with existing documentation, we plan to investigate a more abstract, concept-oriented approach with respect to the code itself. While we hypothesize that automatic documentation generation techniques are sufficiently accurate when describing code, it is possible that they will fail to adequately identify the underlying concepts, thus hindering our ability to identify low-quality comments. However, our previous work suggests that the aforementioned natural language similarity techniques can be effective at comparing human-written documents with source code if care is taken to systematically extract the relevant information from the code [28].

Finally, there are two distinct types of low-quality comments: those that offer incomplete descriptions of the code [41] and those that incorrectly describe the functionality of the code [34]. We hypothesize that incorrect comments may be more detrimental to the overall understandability of a system than comments that are incomplete. Differentiating between the two groups is simple; conceptually, the first type of comment would have a subset of the concepts that the code exhibits while the second type of comment would have a disjoint set of concepts when compared with those present in the code. We propose to study the relative frequency and severity of both types of comments to get a better understanding of the current state of real-world systems and the problems they face with respect to documentation.

4 Proposed Experiments

In this section, we outline our proposed experimental methodology for each of the three proposed research areas. While each area of research directly investigates and improves upon a distinct maintenance task, success in all areas will help to reduce the overall cost of the entire maintenance process.

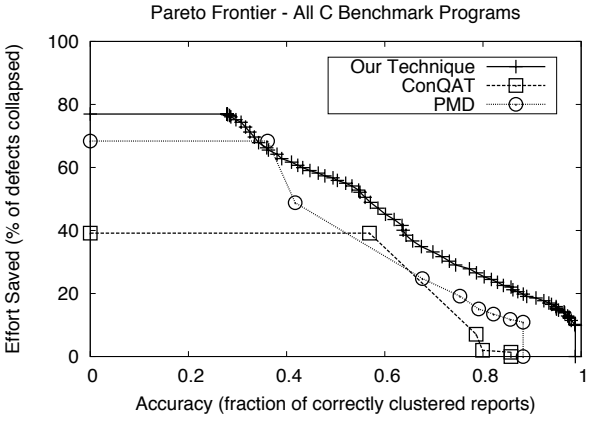

Figure 3: Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate "effort savings" resulting from clustering for C benchmark programs.

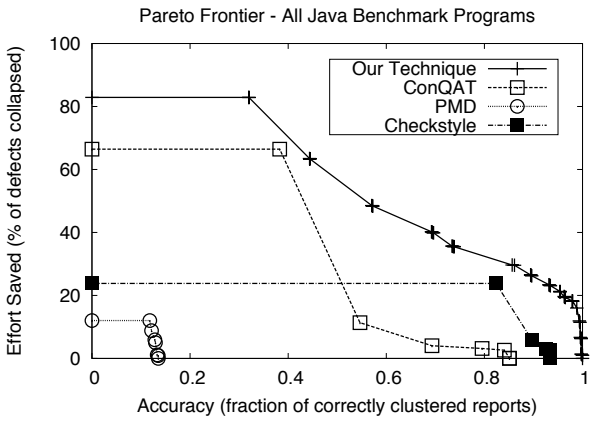

Figure 4: Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate "effort savings" resulting from clustering for Java benchmark programs.

4.1 Experiments: Clustering Duplicate Automatically-Generated Defects

The goal of clustering duplicate automatically-generated defect reports is to reduce the cost of report triage and bug fixing by allowing related defects to be handled in similar ways. Clustering duplicate defects thus has two incomparable goals: cluster accuracy (the internal relatedness of the defect reports in the produced clusters) and the maintenance effort saved from triaging and even fixing clustered defects in similar manners. Enforcing high levels of accuracy will necessarily decrease the size of the resulting clusters and thus reduce the time savings. We thus use both cluster accuracy and effort saved as evaluation metrics and define success as achieving higher effort savings, at 75% accuracy, than existing techniques. Additionally, we hope to be able to output perfectly accurate clusters, regardless of effort savings, to show that our similarity model adequately identifies related automatically-generated defect reports.

Figure 3 and Figure 4 present our preliminary results in terms of Pareto frontiers showing the tradeoff between cluster accuracy (on the x axis) and maintenance effort saved (on the y axis). We ran two state-of-the-art static bug finding tools, Coverity Static Analysis [7] and Finbugs [30], on 14 programs using both C and Java, comprising more than 14 million lines of code. We compare a preliminary implementation of our technique with the most closely related state-of-the-art duplicate detection: code clone detectors. By using the implicated defective code for each report as input to these tools, they can be used to measure defect report similarity and ultimately cluster similar reports in the same fashion as our technique. Our preliminary work outperforms all baseline code clone techniques at almost all levels of cluster accuracy for both C and Java programs.

We also wish to test the quality of clusters produced by our tool with respect to human opinions. Defect report similarity is an inherently human judgment intended to inform a human process. As such, we wish to verify that developers do, in fact, agree with our clusters and would thus benefit from being provided such information. We performed a preliminary investigation by presenting 12 developers with examples of clusters produced by our tool when using a cutoff of 90% internal cluster accuracy. We found that humans found our technique's clusters to be sufficiently related 99% of the time, suggesting that such an approach would be useful in practice.

4.2 Experiments: Facilitating Program Repair

We hypothesize that by developing a fitness function with high fitness-distance correlation we will be able to evolve repairs for more bugs and do so more efficiently. We will evaluate the new fitness function on the

largest, most recent set of established real-world bugs used to test automatic program repair techniques [42]. `GenProg` was previously able to fix 55 bugs out of a total of 105, given a set period of time and resources. Using our newly developed fitness function as part of `GenProg` with the same experimental criteria as in previously published work, we hypothesize we will be able to not only fix more than the previously-repaired 55 bugs, but may take fewer generations (or fitness evaluations) on average to evolve existing fixes. The evaluation metrics for this line of research are thus the number of additional bug fixes realized by our technique and the reduction in time required to find all generated fixes. Additionally, for any bugs fixed in both evaluations, we will qualitatively compare patches evolved using both fitness functions to examine trends in the types or number of mutations used by each.

We further hypothesize that there may be a conceptual divide between bugs that were easily fixed in the past (i.e. those fixed by previous instantiations of `GenProg`) and those that we were previously unable to fix given ample time and resources. Humans employ different strategies to fix different types of bugs: simple bugs can often be fixed quickly with a single code change, while more complex bugs often require larger fixes and additional tools like run-time debuggers or static analysis tools. This conceptual split in bug fixing strategies could conceivably extend to automatically-generated patches as well. We will explicitly test for such a situation by building a model using separate data sets comprised of both types of bug and measuring the relative difference in the statistical importance (using an ANOVA, for instance) of test cases and invariants across different types of defects.

This line of reasoning admits an even finer granularity of investigation into the generality of various features (e.g. test cases and invariants) as they relate to different bugs. We hypothesize that for a given program, such features will generalize across bugs — that is, a test case that is instrumental in fixing one bug will also be important when fixing another. While ultimately this hypothesis can be implicitly tested by the tool's ability to fix additional bugs in practice (generality of features across bugs is crucial to the success of our proposed fitness function), we also plan to explicitly measure the variance in the test cases passed for various program variants across different bugs. Ongoing work in this area is attempting to generalize fixes by developing and enforcing templates for similar classes of bugs. Similarly, gaining a better understanding of how different programs and types of bugs relate with respect to fitness functions may help to inform future improvements to the `GenProg` framework.

4.3 Experiments: Ensuring Documentation Consistency

We wish to identify inconsistent and incomplete comments to ensure long term system quality through accurate documentation. We will first characterize the extent and nature of the problem by manually annotating comments found in large, industrial programs. Others have previous studied the frequency with which code and documentation changes are submitted simultaneously [25, 40], but we know of no efforts to quantify inconsistent and incomplete comments in practice.

To evaluate the effectiveness of our identification technique we will employ the F-measure metric, commonly used to evaluate information retrieval tasks, that favors simultaneously high precision and recall [66]. The main criteria for success for such a technique is an F-measure of over 0.75 when attempting to identify both inconsistent and incomplete comments.

We also propose a pair of human studies designed to both validate our categorization of real world comments and test the effects of different comments in practice. Each study is explained in detail below.

1. We first wish to test our technique's ability to identify incomplete or inconsistent comments by investigating the overlap between the concepts identified by our technique and those identified by humans. To this end, we will split human participants into two groups, asking both to identify incomplete or inconsistent comments and to manually fix them. One group will be given suggestions, produced by our technique, of incomplete or inconsistent concepts to guide them while the other will have to rely only on their intuitions. We hypothesize that the treatment group aided by our technique will identify at least as many missing and inconsistent concepts as the control participants do (i.e. those in the group that do *not* receive our suggestions).

2. The second proposed human study will attempt to show (1) that our tool is as accurate as humans when identifying incomplete or inconsistent comments while also showing (2) that humans find comments that are updated using suggestions from our technique to be of higher quality than those updated using only human intuition. We will present participants with snippets of code and their associated comments that fall in to the following categories:

   (a) An original, consistent, and complete comment as it appears in actual source code.

   (b) An *incomplete* original comment, as it appears in the actual source code, that fails to adequately describe all relevant concepts in the associated code.

   (c) An *inconsistent* original comment, as it appears in the actual source code, that incorrectly describes the concepts in the associated code.

   (d) An *incomplete* comment that has been manually annotated by a participant in the first human study with the help of our technique's suggestions.

   (e) An *incomplete* comment that has been manually annotated by a participant in the first human study using only their own intuitions.

   (f) An *inconsistent* comment that has been manually annotated by a participant in the first human study with the help of our technique's suggestions.

   (g) An *inconsistent* comment that has been manually annotated by a participant in the first human study using only their own intuitions.

   Participants will be asked to first classify each comment as *satisfactory*, *incomplete*, *inconsistent*, or both *incomplete and inconsistent*. Additionally, they will be asked to rate the quality of each comment on a 5-point "Likert" scale [61].

   The controlled nature of this second human study permits us to to test several hypotheses. First, we hypothesize that the use of our tool and the missing or inconsistent concepts it identifies when updating comments will lead to more satisfactory and high-quality comments overall. Testing this hypothesis requires comparing participants' answers for categories (d) and (f) with those from (e) and (g). Additionally, we wish to test the hypothesis that our technique agrees with humans' judgments about inconsistent and incomplete comments as often as they agree with one another. We will compare our technique's output with the human classifications from the first human study as well as those from categories (a), (b), and (c) from this study.

Through the two proposed human studies we hope to meet two specific success criteria: (1) that our comment quality identification technique is at least as accurate as humans at identifying low-quality comments and (2) can also aid in the process of manually updating comments by suggesting inadequately or incorrectly described relevant concepts.

5 Background and Related Work

Clustering Duplicate Automatically-Generated Defect Reports. The large and complex nature of contemporary software systems causes products to be shipped with unknown defects. Many bug finding tools have been developed in recent years to attempt to alleviate this problem [5, 6, 7, 30, 45, 48]. These tools effectively reduce the time it takes to expose defects, but the problem of then triaging and fixing the resulting defects remains a largely manual process. Additionally, several of the projects we ran the Coverity Static Analysis and Findbugs tools on yielded thousands of defect reports, which presents a substantial maintenance cost given previous evidence that developers for real world systems struggle to keep up with even hundreds of defect reports in practice [29].

Duplicate manual defect reports and duplicate code have long been recognized as important issues in software engineering (e.g., [8]). While spurious manual defect reports are an obvious hindrance to the development process, it has been shown that duplicated or semantically related source code also leads to higher

defect densities and thus additional developer effort throughout the maintenance process [39]. There are many tools designed to find code clones for the purpose of removing or refactoring them to aid in future development [14, 18, 51]. Automatic techniques have also been developed to eliminate duplicated human-created bug reports, thus saving developers effort throughout the maintenance process [33, 60, 68]. Human-reported defects often contain a natural language description of the defect in question and optionally a stack trace or automated error output. By contrast, automatic defect detection tools generally produce mostly semantic, code-centric data when identifying potentially buggy statements in the code directly. Techniques attempting to detect manually-created duplicate reports generally focus on matching natural language information while our technique focuses more on the semantic similarities between different pieces of code and thus they are not directly comparable. While duplicate detection has been studied comprehensively as it relates to both source code and manual defect reports, there is a notable lack of research in this area with respect to static analysis results.

Facilitating Program Repair. Localizing and fixing defects is a notoriously difficult and time consuming part of the software maintenance process. Many forms of automated program repair have been developed to reduce this cost [35, 50, 70, 71]. Evolutionary approaches, like the `GenProg` tool, have proven effective at cheaply fixing real world bugs, taking as input only the source code and a test suite that characterizes both the desired program behavior and the bug in question [42]. Borrowing from genetic principles, this technique evolves repairs by making systematic changes over time and promoting only those changes that exhibit desirable qualities with respect to functionality (measured as *fitness*). Previous work has examined the concept of fitness-distance correlation, noting that while such a concept is easily observable for problems with known solutions, it may not generalize across problems and is very difficult to measure for unsolved problems [38]. This work also concludes that measuring the distance between two program variants is most effective if it is based on the types of mutations being used by the search algorithm. This insight inspired the proposed solution of measuring program variant edit distances based on the mutations they encompass. The proposed work further attempts to better define which qualities should contribute to a program variant's *fitness* to increase fitness-distance correlation and thus fix more bugs, more efficiently.

The `GenProg` framework was recently redesigned to minimize memory usage by shifting the way in which program variants and patches are represented [43]. This shift in implementation consequently resulted in a shift in how fitness is viewed. Specifically, the new patch representation describes a program variant in terms of the statement-level changes exhibited when compared with the original program. In this sense, an arbitrary program variant can be described simply by a string of mutations. This representation naturally lends itself to measuring the fitness-distance by computing some notion of edit distance between the string representation of a given program variant and that of known fixes.

Fitness functions have been studied extensively in an effort to improve evolutionary search tasks. Most efforts have focused on reducing the time it takes to evaluate the fitness of a given mutant, either by favoring speed over accuracy [36, 52] or using sampling [24, 73]. While the speed of fitness evaluation is a concern for systems with constrained resources, we instead focus on crafting a more precise fitness function to fix previously unpatched bugs. In the area of fitness function precision, Arcuri *et al.* have explored the use of co-evolution to mutate a program and its test suite simultaneously when attempting to fix bugs, effectively strengthening the fitness function throughout the evolutionary process [1]. Very recent work examined the efficacy of using formal specifications to better inform the accuracy of fitness functions in a single-program case study. The conclusion was that using *only* run-time predicates to guide fitness evaluation yielded a smoother signal that, when used as input to an evolutionary search, could potentially aid in fixing more complex bugs for the one small program evaluated [24]. We propose a unified fitness function that combines and improves upon several of these insights with the end goal of fixing more bugs than was previously possible using the existing `GenProg` framework.

Ensuring Documentation Consistency. Adequate documentation is of paramount importance when attempting to understand a software system [21, 49, 62, 72]. One study specifically examining human readability found that the presence of comments caused humans to report extremely high readability while the lack of documentation caused humans to report the most extreme levels of low readability [65]. More recent studies echoed this finding, noting that humans both explicitly and implicitly find that comments

are strongly tied to understandability [11, 26]. Studies suggest that only around half of all methods in real-world systems are commented in practice [34]. Several tools have been developed to generate documentation where none exists [12, 13, 58, 59]. While research suggests that up to 34% of existing comments may fail to completely document the associated code, relatively little work has been done to identify or remedy such situations [55]. The proposed work will specifically tackle this problem by identifying concepts in source code that the associated documentation does not adequately or correctly explain.

A related maintenance concern is the process by which documentation co-evolves with source code. Previous work has shown that, in practice, developers often fail to update comments when making meaningful code changes, even after having been specifically instructed to do so and given tools to aid in the co-evolution process [25, 40]. While the problem of comment consistency throughout the software life cycle has been established, there has only been preliminary work in identifying "inconsistent" comments. Tan *et al.* have proposed two tools, *iComment* and *@tComment*, that infer strict rules from both comments and method parameters and use both static analysis and automatically generated test cases, respectively, to check that the associated code matches the encoded assertions. These heavy-weight techniques correctly identify very small and specific classes of inconsistent comments, but fail to generalize to the majority of inconsistent documentation. By contrast, the proposed work takes a light-weight, generic approach to identifying the set of *all* inconsistent comments.

Identifying both incomplete and inconsistent comments are tasks related to the overall issue of documentation quality, which has been studied in various ways. Schreck *et al.* measure Javadoc documentation quality in terms of both readability and completeness, where a complete comment is defined as one that mentions all semantic entities for which Javadoc has capabilities to describe (e.g. `@param`, `@return`, and `@throws` statements) [55]. Another tool, JavadocMiner [41], is concerned mostly with measuring quality in terms of how well-formed and complete the documentation's English prose is. We believe issues of readability and natural language quality are adequately measured by the aforementioned tools and thus explicitly do not cover it in the proposed work. We do, however, take a more broad definition of comment completeness than previous tools and thus target a more generic notion of low-quality comments.

The presence and quality of documentation in source code has been found to affect defect density. Several studies have shown that lower quality documentation is linked to higher numbers of defects in the related code [32, 41, 55]. Ibrahim *et al.* note that taking into account inconsistent code and comment changes (where a piece of code is modified without updating the corresponding comment) when predicting bugs improves prediction accuracy with statistical significance [32]. Thus, identifying low-quality comments has the potential to decrease defect density thereby increasing system quality over time.

6 Research Impact

We believe that a particular strength of the proposed research is that it can help to address multiple segments of the software maintenance process. We hope to improve both the bug finding and fixing processes while also ensuring long-term system quality through continued automatic documentation maintenance.

6.1 Leveraging Encoded Domain Knowledge

Humans encode specific domain knowledge in the software artifacts they produce by choosing identifier names, writing comments, or using particular coding patterns and structures. This rich source of information is largely overlooked by many existing maintenance processes. For instance, the process of naming functions and variables requires a human to understand the task at hand and then concisely summarize the related entities to describe parts of the code such that other developers can later understand and update the system. The knowledge gained by understanding both the underlying task and related entities in the code is encoded in the resulting identifiers.

Many software maintenance tasks are related to reverse engineering [16, 19, 44, 69] — as an example, bug finding and fixing can be described as the process of understanding and adhering to program specifications

retroactively. The process of reverse engineering can be defined as the act of trying to rediscover the process by which something was created to gain a better understanding of it as a whole. In the context of software engineering, the *process* describes how the software was written and thus understanding this process necessarily requires understanding the human decisions that were made throughout. The domain knowledge encoded in source code is a great source of information about these decisions.

Many formal methods reject the use of human-centric information in favor of more provably-correct processes. While such information is less structured and not universally-standardized in practice (i.e. developers are not *forced* to pick descriptive identifiers), it can still inform maintenance processes when care is taken to extract and process it properly. The proposed work hopes to effectively extract and use human-centric information, showing its utility by improving maintenance tasks.

6.2 Supporting Existing Processes without Adding Complexity

The goal of the proposed work is to reduce the cost of the maintenance process overall. Processes designed to facilitate software development often impose some amount of additional effort; adding a step to the process necessarily incurs a cost. The goal in designing such processes is to ensure that the intended savings outweighs any additional costs with respect to developer efficiency and overall maintenance effort.

Each of the three research thrusts presented in this document is specifically designed to add minimal complexity (both in terms of time-based, algorithmic costs and logistical, developer-based effort) to the existing maintenance processes. In each case, we employ light-weight analyses specifically designed to reduce the burden on the developer. For example, our approach to clustering duplicate defect reports takes as input exactly the output produced by existing static bug finders, requiring no additionally developer effort.

Even when care is taken not to add superfluous costs when attempting to facilitate maintenance tasks, quantitatively measuring such a tradeoff is often difficult because of the amount of variability inherent in existing maintenance tasks. Consider the task of automatic program repair — to precisely compare such a technique with existing processes one has to know how long it takes to manually fix a bug, on average [42]. While some bugs can be located and fixed in a matter of minutes, others can persist through several releases of a product. Comparatively, consider an interactive code search tool designed to quickly locate concepts in source code based on iterative user feedback. While some users may effectively speed up their task throughput using such a tool, others may find that the complexity such a technique adds to their existing toolkit is prohibitive in terms of increasing productivity. Both of these examples show that concretely measuring the cost benefits of maintenance tools is difficult when baselines are not well established.

To address this ambiguity we specifically ensured each area of proposed work is directly comparable to a known or explicitly measurable baseline. When clustering defect reports, we propose to compare the "collapsed" set of defect reports (where each cluster conceptually corresponds to the effort it would take to triage and fix a single defect) with the set of reports originally produced by the static bug finder. Thus, we circumvent the problem of having to quantify the maintenance effort associated with a single defect report by making an argument for the percent reduction of effort overall. Considering our proposed improvements to automatic program repair (which require no additional developer effort) we can compare against existing automatically-generated patches with respect to time and cost. Finally, we will conduct a human study to directly measure the effort and quality implications our low-quality comment identification technique poses when manually updating documentation (see Section 4.3 for more detail). The existence of grounded and direct baselines from which to evaluate our proposed work allows for higher confidence in the resulting conclusions about the effects our techniques have on the maintenance process.

6.3 Summary

By systematically extracting and using latent human-centric information present in software artifacts we propose improvements to three crucial maintenance tasks. Existing techniques often favor provably trustworthy sources of information (e.g. code structure or control flow) over more speculative, but knowledge-rich, forms of information (such as identifier names and natural language comments). The proposed techniques

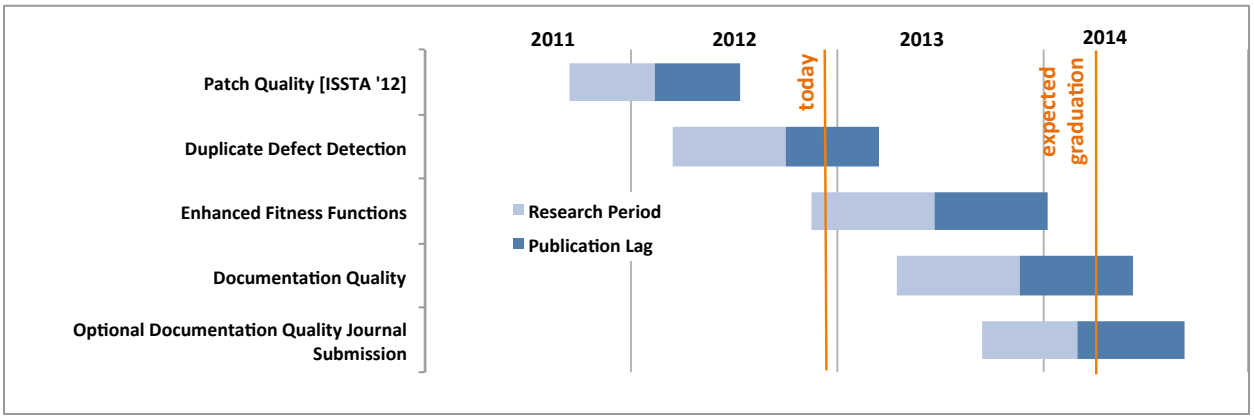

Figure 5: Proposed research schedule.

focus on these often-ignored sources of information to facilitate maintenance tasks and improve the overall maintenance process in practice.

Additionally, the proposed techniques are specifically designed to be lightweight so as to require minimal additional time or developer effort. A theoretically-effective but practically-laborious technique may not gain wide adoption. The maintenance process is already prohibitively expensive when trying to find and fix all bugs — adding additional complexity can yield diminishing returns. By design, our techniques require little or no additional developer effort and aim to save more human and computational time than they consume in practice.

7 Research Plan

The proposed work comprises three main research thrusts for which we hope to publish at least three papers. In the past we have targeted conferences related to software maintenance and software analysis including *The International Conference on Software Maintenance (ICSM)*, *The International Symposium on Software Testing and Analysis (ISSTA)*, and more broadly *The International Conference on Software Engineering (ICSE)*. For the proposed papers we will again consider these venues in addition to *The International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* and *The Genetic and Evolutionary Computation Conference (GECCO)*.

Figure 5 outlines the proposed schedule for the work outlined in this document. Uncertainty in the schedule includes the possibility of an additional journal paper on the subject of documentation quality that includes additional evaluations based on further human studies as well as the potential revision and resubmission of previous papers currently submitted for publication but not directly related this Ph.D. proposal.

8 Summary and Long-Term Vision

The long term goal of our work is to reduce the cost of the maintenance process by developing light-weight, generic analyses that leverage unstructured developer information to improve three crucial processes: bug finding and triage, defect fixing, and persistent system quality assurance. We will evaluate the proposed techniques directly, either against existing state-of-the-art tools or using human studies to measure actual developers' intuitions.

Current maintenance processes are not sufficient to find and fix all bugs efficiently and effectively in practice. To ease part of this maintenance burden we propose improvements to three specific tasks:

- Clustering automatically-generated defects to facilitate bug triage and repair (Section 3.1.1)
- Improving fitness functions to aid in automatic program repair (Section 3.1.2)
- Identifying inconsistent and incomplete comments to ensure long-term system quality (Section 3.1.3)

As the dominant cost throughout the software life cycle, software maintenance plays a crucial role in the success and longevity of a piece of software. We hope to ease the maintenance burden by producing techniques that produce high-quality results (by leveraging under-exploited human-centric information) that can be easily and incrementally adopted (because they are implemented using lightweight analyses).

References

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

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Program Analysis for Software Tools and Engineering*, pages 1–8, 2007.

[3] T. Ball. A theory of predicate-complete test coverage and generation. In *Proceedings of the Third international conference on Formal Methods for Components and Objects*, pages 1–22, 2005.

[4] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, May 2001.

[5] T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Principles of Programming Languages*, pages 1–3, 2002.

[6] M.-C. Ballou. Improving software quality to drive business agility. White paper, International Data Corporation, June 2008.

[7] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. 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.

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

[9] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[10] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

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

[12] R. P. L. Buse and W. Weimer. Automatically documenting program changes. In *Automated Software Engineering*, pages 33–42, 2010.

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

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

[15] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2004.

[16] B. H. Cheng and G. C. Gannod. Abstraction of formal specifications from program code. In *in Proceedings for the IEEE 3rd International Conference on Tools for Artificial Intelligence*, pages 125–128. IEEE, 1990.

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

[18] ConQAT. Conqat. `https://www.conqat.org/`, 2011.

[19] M. D'Ambros, H. Gall, M. Lanza, and M. Pingzer. *Software Evolution*, chapter Analysing Software Repositories to Understand Software Evolution, pages 37–67. Springer Berlin Heidelberg, 2008.

[20] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[21] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[22] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[23] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, 2001.

[24] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, pages 965–972, 2010.

[25] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. WCRE '07, pages 70–79, 2007.

[26] Z. P. Fry, B. Landau, and W. Weimer. A human study of patch maintainability. In M. P. E. Heimdahl and Z. Su, editors, *International Symposium on Software Testing and Analysis*, pages 177–187, 2012.

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

[28] Z. P. Fry and W. Weimer. Fault Localization Using Textual Similarities. *ArXiv e-prints*, 2012.

[29] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated Software Engineering*, pages 34–43, 2007.

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

[31] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, 2006.

[32] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. Controversy corner: On the relationship between comment update practices and software bugs. *Journal of Systems and Software*, 85(10):2293–2304, 2012.

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

[34] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. MSR '06, pages 179–180, 2006.

[35] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Programming Language Design and Implementation*, 2011.

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

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

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

[39] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *International Conference on Software Engineering*, pages 485 –495, 2009.

[40] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.

[41] N. Khamis, R. Witte, and J. Rilling. Automatic quality assessment of source code comments: the javadocminer. In *International Conference on Application of Natural Language to Data Bases*, pages 68–79, 2010.

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

[43] C. Le Goues, S. Forrest, and W. Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, pages 959–966, 2012.

[44] C. Le Goues and W. Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306, 2009.

[45] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, pages 141–154, 2003.

[46] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.

[47] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[48] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, pages 89–100, 2007.

[49] D. L. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, 1994.

[50] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, 2009.

[51] PMD. Pmd. `http://pmd.sourceforge.net/pmd-5.0.0/`, 2012.

[52] K. Sastry and D. E. Goldberg. Genetic algorithms, efficiency enhancement, and deciding well with differing fitness variances. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 528–535, 2002.

[53] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[54] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[55] D. Schreck, V. Dallmeier, and T. Zimmermann. How documentation evolves over time. International Workshop on Principles of Software Evolution, pages 4–10, 2007.

[56] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[57] H. Siy and L. Votta. Does the modern code inspection have value? International Conference on Software Maintenance, pages 281–, 2001.

[58] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. International Conference on Automated Software Engineering, pages 43–52, 2010.

[59] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. International Conference on Software Engineering, pages 101–110, 2011.

[60] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *International Conference on Software Engineering*, pages 45–54. ACM, 2010.

[61] Symantec. A technique for the measurement of attitudes. In *Archives of Psychology*, volume 22, page 55, 1932.

[62] A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 4(3):143–167, 1996.

[63] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or bad comments? */. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, October 2007.

[64] S. H. Tan, D. Marinov, L. Tan, and G. 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.

[65] T. Tenny. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, 1988.

[66] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

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

[68] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *International Conference on Software Engineering*, pages 461–470, 2008.

[69] M. P. Ward. Reverse engineering through formal transformation - knuths "polynomial addition" algorithm. *The Computer Journal*, 37:795–813, 1994.

[70] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, pages 61–72, 2010.

[71] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[72] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *International Conference on Software Engineering*, pages 215–223, 1981.

[73] T.-L. Yu, D. E. Goldberg, and K. Sastry. Optimal sampling and speed-up for genetic algorithms on the sampled onemax problem. In *Genetic and Evolutionary Computation Conference*, pages 1554–1565, 2003.