A Literature Review of Research in Software Defect Reporting

Jonathan D. Strate, *Member, IEEE*, and Phillip A. Laplante, *Fellow, IEEE*

***Abstract*—In 2002, the National Institute of Standards and Technology (NIST) estimated that software defects cost the U.S. economy in the area of $60 billion a year. It is well known that identifying and tracking these defects efficiently has a measurable impact on software reliability. In this work, we evaluate 104 academic papers on defect reporting published since the NIST report to 2012 to identify the most important advancements in improving software reliability though the efficient identification and tracking of software defects. We categorize the research into the areas of automatic defect detection, automatic defect fixing, attributes of defect reports, quality of defect reports, and triage of defect reports. We then summarize the most important work being done in each area. Finally, we provide conclusions on the current state of the literature, suggest tools and lessons learned from the research for practice, and comment on open research problems.**

***Index Terms*—Software defect reporting, software quality assurance, software test automation.**

Acronyms

ACM	Association for Computing Machinery
NIST	National Institute of Standards and Technology

I. Introduction

In a widely cited study, the National Institute of Standards and Technology (NIST) estimated that software defects cost the U.S. economy in the area of $60 billion a year. The NIST study also found that identifying and correcting these defects earlier could result in upwards of $22 billion a year in savings. The gap between cost and savings shows defect identification and reporting to be an essential part of improving the reliability of software.

Software companies usually manage identified defects though defect tracking systems. The data entered into these systems and how it is used has a direct impact on fixing defects. For example, in one study, researchers examined the attributes of defect reports in the defect tracking systems of nine companies and found that most of the data collected did little to contribute to the quality of the report. After recommending and implementing changes to what data is collected, the share of post-release defects due to faulty defect fixes dropped from 33% to 8% [1]. This measurable improvement demonstrates the significance of continued analysis of defect reporting and management in improving the reliability of software.

TABLE I
Source of Papers Reviewed

Conference Proceedings	**Papers**
Published by ACM	29
Published by IEEE Computer Society	51
Published by KSI Press	2
Published by Springer-Verlag	1
Total	**83**
Journal Articles	**Papers**
ACM	
- SIGARCH Computer Architecture News	1
- ACM Trans. on Software Engineering and Methodology	1
- Communications of the ACM	1
IEEE Computer Society	
- Computer	1
- IT Professional	1
- IEEE Transactions on Evolutionary Computation	1
- IEEE Software	1
- IEEE Transactions on Software Engineering	2
Elsevier	
- Applied Soft Computing	1
- Information Sciences	1
- Information and Software Technology	1
- Journal of Systems and Software	2
Springer	
- Empirical Software Engineering	1
- Innovations in Systems and Software Engineering	2
Wiley	
- Journal of Software: Evolution and Process	1
Total	**18**
Other	**Papers**
Book	1
Ph.D. Thesis	1
Technical Report	1
Total	**3**
Total Papers Reviewed	**104**

In this work, we reviewed 104 academic papers on defect reporting published since the NIST report to 2012. To find these papers, we searched the digital libraries of IEEE, Association for Computing Machinery (ACM), Elsevier, Springer, and Wiley. We also conducted a keyword search in Google Scholar to identify noteworthy unpublished papers (Table I).

In the search parameters, we used the term "defect" to refer to any error, fault, failure, or incident as defined in the IEEE standard glossary. We used the term "bug" as a search term, but captured related papers only if it was clear that the authors intended to use the term interchangeably with defect.

In choosing the papers, we kept the scope to research targeting the creation of defect reports to when they are triaged. This scope included ways to automatically detect and fix defects, the content and quality of manually-submitted defect reports, and the automatic and manual triaging of defect reports.

Manuscript received June 03, 2012; revised November 02, 2012; accepted November 26, 2012. Date of publication April 29, 2013; date of current version May 29, 2013. Associate Editor: J.-C. Lu.

The authors are with the Pennsylvania State University, Malvern, PA 19355 USA (e-mail: jds361@psu.edu; plaplante@psu.edu).

Digital Object Identifier 10.1109/TR.2013.2259204

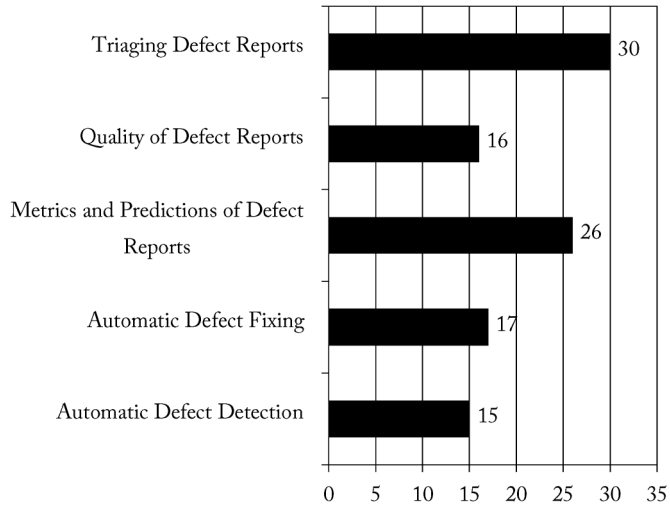

Fig. 1. Number of papers reviewed in each area.

We looked at the commonalities of the research in these areas and found five branches of research regarding defect reporting. We then categorized each paper to one of the five areas (Fig. 1).

1) **Automatic Defect Fixing**—Research in this area looked at ways to evolve code to automatically fix defects.
2) **Automatic Defect Detection**—Research in this area concentrated on the different approaches for identifying defects such as defects caused by changes, looking at past defects as an indicator of future defects, analyzing complex software components, churn and entropy, and identifying the number of defects in a software release.
3) **Metrics and Predictions of Defect Reports**—Research in this area used data from defect reports to make predictions for how long the defect will take to get fixed, the severity, the complexity, the likelihood of a fix, and other attributes that assist in managing a defect tracking system.
4) **Quality of Defect Reports**—Research in this area looked at what makes a good defect report and how to improve defect reporting systems. The motivation in this research is that the quality of a defect report often determines how quickly it will be fixed.
5) **Triaging Defect Reports**—Research in this area was concerned with automatic assignment of defect reports, detecting and addressing duplicate defect reports, and distinguishing valid defect reports from enhancement requests and technical support.

This work presents the contributions to defect reporting in each area since the NIST report. We conclude by noting the advancements in each area of research, and recommend emerging tools developed in academia that apply to industry practice.

II. Automatic Defect Fixing

The process of identifying and fixing defects is a largely human task. Unit testing can assist when new code breaks previously functioning code, but keeping the tests up-to-date can be challenging as well. The area of automatic defect fixing identifies more automatic approaches that propose fixes to unit tests and fixes to found defects.

A. Applications of Genetic Programming

In software engineering, debugging is mostly a manual process. A fully automated approach to discover and fix defects is becoming more feasible with research focusing on applying the theories of genetic programming.

Arcuri and Yao proposed an approach to automatically fix defects based on co-evolution [2]. The idea is to evolve both programs and test cases at the same time. The user would provide both a program containing defects and formal specification for the program [3]. The output would be an automatically-fixed program. A research prototype called Java Automatic Fault Fixer (JAFF) was created to test this approach [4].

Weimer, Nguyen, Le Goies, and Forrest devised a way to evolve program variants of a fault until a variant is found that both avoids the defect and retains functionality [5]. The advantage of this approach is that it can be applied to programs without formal specifications. Further research was done to improve these methods by focusing on particular modules to limit the search space complexity [6], [7]. These methods were applied to the Siemens commercial software suite by Debroy and Wong. Their mutation technique was shown to fix 18.52% of the faults automatically [8].

Tangential to defect fixing, the idea of using genetic programming to improve non-functional properties like execution time or memory usage was proposed by White, Arcuir, and Clark [9]. In their proposal, non-obvious improvements are made to non-functional properties of code.

B. Crossovers With Search Based Software Engineering

Part of automatically fixing defects involves searching code for possible defects. The current state of the practice generally involves a human-based search function with the mass generation of defect reports. However, the goal of Search Based Software Engineering (SBSE) is to move this engineering problem to a machine-based approach [10]. Evolutionary computation has fueled SBSE, and can be used anywhere a suitable fitness function can be defined [11].

Nainar and Liblit proposed the concept of "Adaptive Bug Isolation" where a search on the control-dependence graph eventually zeros in on defects. They suggested some heuristics that can be used for the search and gave some metrics showing their method only adds 1% to the performance overhead in large applications [12].

C. Tools for Automatic Defect Fixing

Using the theories of evolutionary computing and SBSE, several tools were developed by researchers.

- **Coevolutionary Automated Software Correction**—Developed by Wilkerson and Tauritz, this method automates the cycle of software artifact testing, error location, and correction [13]. The approach is co-evolutionary, where software artifacts and test cases evolve together continuously. The goal is the evolution of test cases to better find defects, and the software to better behave according to the specification of the test cases. The result is the continuous improvement of code due to this "evolutionary arms race."

- **AutoFix-E and AutoFix-E2**—The AutoFix-E tool automatically generates and validates fixes for software faults. Using this tool, researchers were able to propose successful fixes to 16 of 42 faults found in the Eiffel libraries [14]. That work was continued to more broad applications in AutoFix-E2 [15]. Both versions rely on contracts in the code to propose successful fixes.
- **ReAssert**—ReAssert was developed by Daniel, Jagannath, Dig, and Marinov to repair failing unit tests [16]. The tool suggests repairs to the code that would allow the failing test to pass, leaving the developer to choose the best repair. The tool was later enhanced by Daniel, Gvero, and Marinov to add symbolic test repair [17]. This enhancement uses the Pex symbolic execution engine as the algorithm to suggest fixes.
- **GenProg**—Described by Le Goues, Nguyen, Forrest, and Weimer, GenProg was designed as an automated method for repairing defects in software that lacks formal specifications [18]. The algorithm is a form of genetic programming that evolves program variants based on the required inputs and outputs of test cases. To ensure the evolved code change is as minimal as possible to the original code, structural differencing algorithms and delta debugging are employed before the variant is committed.

III. Automatic Defect Detection

Like defect repair, the work of identifying defects is also a significant manual process. The goal of automatic defect detection is to automate this process to identify more defects in a shorter amount of time.

A. Approach Analysis

Some common approaches to automatic defect detection include analyzing change metrics for defects caused by changes [19], looking at past defects with the idea they could indicate future defects [20], analyzing the more complex software components with the idea they are more error prone due to the difficulty in changing them [21], and using churn and the complexity of changes (entropy) from source code metrics [22].

D'Ambros, Lanza, and Robbes analyzed these approaches and proposed a benchmark engineers could use to produce an easy comparison between them [22]. Approaches are based on both explanative power and predictive power, with both powers computed mathematically based on the metrics used and reported from each method. Another analysis was performed by Peng, Wang, and Wang on the most appropriate classification algorithms used with the different methods [23].

Aside from these approaches, Williams and Hollingsworth proposed a method to use the change history of source code to assist in the search for defects [24]. While static source code checkers are currently available, this approach uses the historical data from the change history to refine the results.

DeMott, Enbody, and Punch found that some approaches take so long it is a challenge to integrate them continuously in the development pipeline [25]. Their work uses distributed fuzzers to achieve high-output needing only manual intervention to review proposed defects.

B. Software Inspection

Software inspection is a method of detecting faults in phases of the software life cycle. Predicting the number of defects is an important metric for project managers to assess when to stop testing. Petersson, Thelin, Runeson, and Wohlin reviewed the improvements in the capture-recapture method of defect detection between 1992 and 2002 [26]. Chang, Lv, and Chu noted that the number of defects found in the capture-recapture model is generally overestimated. They proposed using a method based on Maximum Likelihood Estimator that performed two inspection cycles to increase the accuracy of estimation [27]. Bucholz and Laplante extended the capture-recapture model from the pre-release stages to the post-release stage by including defect reports [28].

Zimmermann, Premraj, and Zeller showed that a combination of complexity metrics can predict defects, suggesting that the more complex code it has, the more defects it has. [29]. Nagappan, Ball, and Zeller developed a regression model to predict the likelihood of post-release defects [30]. Fenton, *et al.* found a way to form this prediction using Bayesian Networks in lieu of regression [31].

C. Tools for Automatic Defect Detection

Several tools were developed by researchers to assist in the link between defect reports and the defects themselves.

- **Linkster**—Linkster was designed to reverse-engineer links between code repositories and big databases [32]. The motivation arose out of the need to identify the links between defect databases and program code repositories to give data to software defect analysts.
- **BugScout**—BugScout uses existing defect reports to narrow the search space for the defect-prone code and point developers to the likely place in code that is causing the defect. The initial evaluation of BugScout found that it identified the correct area of code 45% of the time [33].

IV. Metrics and Predictions of Defect Reports

As software artifacts, defect reports are a valuable source of information. We can show quantitatively that more ambiguous requirements lead to more defects. Using statistics, predictions can be made for how long it will take to fix a defect, the severity, the complexity, and if a defect is likely to be fixed or reopened. In addition, metrics can be shown for how developers and users collaborate.

A. Gathering Data

Research was done to show that defect reports can be traced back to ambiguous requirements. Wasson, Schmid, Lutz, and Knight gave some recommendations on dealing with natural language issues in requirements that lead to defects later [34]. Some of the issues they found could be traced back to common linguistic devices that represent under-specification, such as usage of the words "like" and "some."

D'Ambros, Lanza, and Pinzger conducted novel research on visualizing a defect database. Their motivation was that visualization is a useful way to display large amounts of data, such as the data available in a defect tracking database. They proposed a "system radiography view" that shows how open defects are

distributed in the system and over time, as well as a "bug watch view" that models the defects affecting a singular component during one time interval to detect the most critical issues [35].

Ayewah and Pugh found that some changes linked to defect reports do more than just fix the defect; they enhance the code [36]. However, this enhancement is one of the issues identified that makes tracing the origins of defects a challenge [37]. Rastkar, Murphy, and Murray found that it can take significant time for developers to read through the contents of a defect report to get an idea of the issue, so automatically generating summaries by parsing the text of the report can be useful [38].

Chilana, Ko, and Wobbrock found that there is sometimes a difference between the developer's intent and what the users expect. As a result, they were able to classify defects as either violations of users' own expectations, of specifications, or of user community expectations.

They also connected this classification to the fix-likelihood, and found that those defects that were violations of user community expectations had a higher chance of being fixed than when defects were considered to be violations of users' own expectations [39].

B. Statistical Predictions

By making predictions, software developers can better prioritize their tasks. However, depending on the point in a project's life cycle, the defect prediction models can be reliable or unreliable [40]. We break the research down to the different predictable attributes of defect reports.

- **Time to fix**—Weiss, Premraj, Zimmerman, and Zeller claim to be able to predict the fix time within the hour by using a sufficient number of defect reports [41]. This approach can be improved with the inclusion of post-submission defect report data [42]. However, research found no correlation between defect-fix likelihood, defect-opener's reputation, and the time it takes to fix a defect [43]. Marks, Zou, and Hassan studied fix-time along three dimensions: the component of the defect, the reporter, and the description. They found that they could correctly classify defects 65% of the time for Eclipse and Mozilla [44].
- **Severity**—Lamkanfi, Demeyer, Giger, and Goethals devised an algorithm that analyzes the text description of defects to determine severity. With a training set of roughly 500 reports per severity, they can predict the severity with precision, and recall around 70% on the Mozilla and Eclipse projects [45]. Continuing on that research, it was found that Naïve Bayes Multinomial performs better than other algorithms for predicting the severity of a reported defect [46].
- **Complexity**—Using the estimated fix time of the defect, a complexity cluster can be created using k-means defining defects of low, medium, and high complexity [47].
- **Popularity**—This dimension is researched to supplement other dimensions. Bacchelli, D'Ambros, and Lanza found that the number of e-mails generated that discuss a defect is related to more problematic issues, and this popularity metric can improve the prediction of other metrics by up to 15% [48].
- **Which defects get fixed**—One large study of Microsoft Windows focused on the characteristics that affected which defects get fixed. The factors reduced to defects reported by users with better reputations, and defects handled by people in the same group working in the same geographical location. A statistical model was built based on those factors that was able to achieve a 68% precision on predicting Windows 7 defect fixes [49].
- **Which defects get reopened**—Researchers show we can predict the probability a defect will be reopened by looking at the comment and description text, the time it took to fix the defect, and the component in which the defect was found. This prediction model achieves a 62.9% precision, and 84.5% recall, on the Eclipse project [50].

C. Developer and User Collaboration

Defect reports provide data into how developers collaborate. Analyzing this collaboration can give useful insight into both software quality assurance and the development process. Developer collaboration data can be mined from version control logs, and research is showing that supplementing this mining with data from defect reports can enhance the information gathered. It can also be used later by expert systems for automatic assignment during triage. However, this description gives only an incomplete picture, and more can be added to defect reports themselves to improve the picture at the end [51]. Adding the solution originator and solution approver to defect reports helps better establish who is developing when analyzing development artifacts [52]. In addition, although it might seem that in small teams face-to-face communication would trump all communication, it was found that defect repositories are still a fundamental communication channel [53].

Expert developers can be derived from the information in defect reports. Anvik and Murphy proposed a method to mine defect repositories for information about which developers have expertise in the implementation of a code base [54]. It was also found that users who actively participate in the resolution of defects with the developers helps the defect make progress [55]. Their ongoing participation helps developers get clarification on what the user expectations are, and what the current implementation provides.

Ko and Chilana found that open defect repositories, contrary to conventional wisdom, are useful for deriving info from the masses in only a few cases. Most useful defect reports come from a small group of experienced, frequent users, with most users reporting issues into an open defect repository that could be resolved with technical support [56].

D. Tools for Metrics and Predictions of Defect Reports

Several tools were developed to assist with metrics and predictions.

- **RETRO**—The Requirements Tracing On-Target (RETRO) tool addresses the problem of tracing textual requirements to related textual defect reports. Evaluation of the tool shows a recall of between 70% and 85%, and a precision of between 85% and 99%, on datasets of a NASA scientific instrument [57].

- **SEVERIS**—The Severity Issue Assessment (SEVERIS) tool was motivated by the necessity to evaluate the severity of defects discovered during testing. SEVERIS is based on text mining and machine learning on existing defect reports. Preliminary tests found it to be useful with data from the NASApsilas Project and Issue Tracking System (PITS) [58].
- **BugTrace**—BugTrace was developed as a method to analyze patches to create links between defect reports and source code. It mines and merges the data between a CVS repository and a Bugzilla repository, and from the evaluation produced few false positive and few false negatives for the 307 defects in the test data [59].

V. Quality of Defect Reports

Defects can be fixed more quickly and accurately when the reports themselves are more useful to developers. Surveys were done to discover what makes defect reports useful, and suggestions were proposed on how to make defect reports more useful for developers and users.

A. Surveying Developers and Users

In one survey of the quality of defect reports in Eclipse, developers said that those reports that included stack traces were most useful, while those containing erroneous, incomplete information were the least useful in addressing the defects [60].

The researchers concluded that incomplete information slows down the defect fixing process, constant user involvement was essential for successful defect reports, and the associated defect reporting tools could be improved [61].

Another survey found that the steps to reproduce the defect, and the observed behavior from that reproduction, are the most important aspects of defect reports. However, many defect report creators lack this technical information, and the authors recommended finding ways to automate this process [62]. Other research recommends adding that automation into new defect tracking systems.

Specifically, any new defect tracking systems should provide contextual assistance and reminders to add missing information to users who are reporting defects [63]. To assist in that automatic text mining of defect reports, Ko, Myers, and Chau reported that analyzing the titles of defect reports found 95% of the noun phrases referred to visible software entities, physical devices, or user actions [64].

B. Improving Defect Reports

Weimer found that defect reports with patches were three times more likely to be addressed [65]. He proposed using fuzzy logic to construct a patch along with each defect report.

Castro, Costa, and Martin acknowledged better defect reports include inputs that make the software fail, but many vendors will avoid including those inputs at the risk of collecting personal information from users. They provided a mechanism to measure the amount of information a user may consider private in an error report, and a way to submit the defect with almost all that private information eliminated. This redaction is done by using a Satisfiability Modulo Theory (SMT) solver to compute a new input generating the exception without using the original input that may contain personal information [66].

Dit, Poshyvanyk, and Marcus noted that the user comments of defect reports can degenerate and lose context with the original defect [67]. Therefore, it is useful to measure the textual coherence of the comments. Using Information Retrieval (IR) techniques to measure comments, they found comments with a high textual coherence were associated with better defect reports [68].

Zimmermann, Premraj, Sillito, and Breu proposed four ways to improve defect tracking systems: first by collecting stack traces, second by helping users provide better information, third by using something like automatic defect triage to improve the process, and fourth by being very clear with the users on what is expected by the developers in defect reports [69]. Schroter, Bettenburg, and Premraj contributed empirical evidence that stack traces help developers fix defects [70].

Nguyen, Adams, and Hassan warned about the bias in pulling datasets of defect repositories for research, noting that assumptions exist like assuming they contain only defect reports when really they also contain enhancements [71].

Sun found that, of invalid defect reports, 46% are invalid because of errors in testing, while 28% are invalid because of a misunderstanding on functionality [72]. For the 46% of errors in testing, many were because of lack of detail in the test cases, or the sequence of actions in them were followed incorrectly.

Li, Stålhane, Conradi, and Kristiansen studied the attributes of defect reports in the defect tracking systems of nine Norwegian companies, and found that most of the data collected did little to contribute to the quality of the defect report. For example, many of the reports contained incomplete data, where 20% or more of the data for an attribute was left blank.

Mixed data, where several attributes were assumed to be combined, were also found to be troublesome in defect reports. For example, where it would be assumed the source of the defect would be included in the description. In one company, they added attributes for effort (classified as either quick-fix or time-consuming), fixing type, severity, trigger, and root cause. As a result, after 12 months, the share of post-release defects attributable to faulty defect fixes had dropped from 33% to 8% [1].

C. Tools to Improve Defect Report Quality

There are several tools to assist in the improvement of defect reports.

- **CUEZILLA**—CUEZILLA was developed to measure the quality of defect reports and recommend to submitters what can be improved before submission. This work was motivated by research that indicated that, while developers find the steps to reproduce an error, stack traces, and associated test cases as the most useful artifacts in fixing a defect, users without technical knowledge find these data difficult to collect. On a training sample of 289 defect reports, the researchers' tool was able to predict the quality of 31% to 48% of defect reports [73].
- **GUI Monitoring and Automated Replaying**—Herbold, Grabowski, Waack, and Bünting developed a proof-of concept implementation of a GUI monitoring system that generates monitored usage logs to be replayed when a defect

occurs. A way to monitor the GUI usage that led to identifying a defect is useful to the software developer to fix the defect, and it was found that integrating this proof-of-concept tool into a large-scale software project took little effort [74].

VI. Triaging Defect Reports

Triaging defect reports can take time away from hours spent fixing code. Therefore, automatic ways to assign defects, classify them, or detect duplicates leads to less for a developer to do to investigate a defect before a defect fix can be implemented.

A. Methods for Automatic Assignment

Laplante and Ahmad noted that manual defect assignment policies can affect customer satisfaction, both the morale and behavior of employees, the best use of resources, and software quality and success [75]. Such a strong impact makes the research in the area of assignment beneficial. We present several active areas of research in the area of automatic assignment of defect reports.

- **Text Categorization**—Čubranić and Murphy presented a text categorization technique to predict the developer that should work on the defect based on the submitter description. In a collection of 15,859 defect reports from a large open source project, their approach can correctly predict 30% of the assignments using supervised Bayesian learning [76].
- **Machine Learning**—Anvik, Hiew, and Murphy applied a machine learning algorithm to suggest a number of developers in which a defect report would be assigned. On the Eclipse project, this method achieved a precision level of 57%. On the Firefox project, this method achieved a precision level of 64% [77]. The machine learning approach can also create Recommenders. Recommenders are algorithms with information about previously fixed defect reports as a model of expertise [78]. When applying the machine learning recommender approach, Anvik and Murphy had a precision between 70% and 90% over five open source projects [79]. Zou, Hu, Xuan, and Jiang found that these results suffer from large-scale but low-quality training sets. They proposed reducing the training set size. In experiments on the Eclipse project, they found that, with a 70% reduction in words and 50% reduction in defect reports, they could reduce the number of defect repair assignments without reducing their accuracy [80]. Bahattacharya, Neamtiu, and Shelton applied several dimensions of machine learning (classifiers, attributes, and training history) to increase prediction accuracy. Using this method, they were able to achieve 86% accuracy on Mozilla and Eclipse [81].
- **Markov Chains**—Jeong, Kim, and Zimmermann created a graph model based on Markov chains. This model creates a defect tossing history, where tossing refers to the reassignment of a defect to another developer. Using this method on 445,000 defect reports, this model reduced defect tossing events up to 72%, and prediction accuracy by up to 23%, compared to other approaches [82].
- **Support Vector Machine (SVM)**—Lin, Shu, Yang, Hu, and Wang chose to implement an assignment using SVM because it is effective in dealing with high dimensions of data. Their results were close to the precision levels achieved by the machine learning approach used by Anvik, Hiew, and Murphy [83].
- **Term-Author-Matrix and Term Vectors**—Matter described a method for automatic assignment by creating Term-Author-Matrices, associating developers with terms that surface in the defect reports. From there, term vectors are developed from each defect report, and matched with the correct authors conducting the assignment. Using Eclipse, this method achieved a 33.6% top-1 precision, and 71.0% top-10 recall [84].
- **Information Retrieval**—Matter, Kuhn, and Zierstrasz proposed a method to locate source code entities from the textual description of the defect report, and then to search relevant commits in source control. They achieved 34% top-1 precision, and 71% using Eclipse [85]. Kagdi, Gethers, Poshyvanyk, and Hammad also proposed a method and demonstrated accurately assigning defect reports between 47% and 96% of the time [86].

B. Algorithms to Classify Defect Reports

Podgurski, *et al.* proposed an automated method for classifying defects to assist in triage. The classification was to group defects together that may have the same or similar causes. By examining the frequency and severity of the failures, more informed prioritization could take place. They found their method to be effective for self-validating failures in the GCC, Jikes, and javac compilers [87].

By using alternating decision trees, Naïve Bayes classifiers, and logistic regression, Antoniol, *et al.* was able to classify issues into either defects or non-defects consisting of enhancement requests or other issues. They were able to correctly classify issues between 77% and 82% of the time for Mozilla, Eclipse, and JBoss [88].

Gegick, Rotella, and Xie researched an algorithm to use text mining on natural language descriptions to classify if a defect is also a security risk. On a large Cisco software system, their model was able to accurately classify 78% of the security-related defect reports [89].

Xuan, Jiang, Ren, Yan, and Luo proposed a semi-supervised approach using a Naïve Bayes classifier and expectation maximization. They were able to out-perform previous supervised approaches for defect reports of Eclipse [90].

Xiao and Afzal created an algorithm using genetic algorithms as a search-based resource scheduling method for defect-fixing tasks. This approach is useful for changing the parameters, such as the number of developers or relaxing a delivery date, and can show how many additional defects should expect to be fixed given different resources or timelines [91].

Khomh, Chan, Zou, and Hassan proposed triaging new defects into crash-types. This approach takes into account many factors, including the severity, frequency, and fix time proposed. They found their method using entropy region graphs to be more effective than the current triaging used by the Firefox developers [92].

C. Detecting and Addressing Duplicates

In studying the factors that lead to duplicate defect reports, it was found that the number of lines of code and the life-time of the project do not impact the number of duplicates. Similarly, the number of defect reports in the repository is also not a factor that causes duplicate defect reports [93]. However, conventional wisdom indicated duplicate defect reports slow down the defect fixing process with resources needed to identify and close duplicate defects [94]. This belief was refuted in another survey to determine if duplicate defect reports are considered harmful. The data collected revealed that those duplicate defect reports provide useful data collectively, and should be combined [95].

Wang, Zhang, Xie, Anvik, and Sun found that duplicate detection could be improved with both natural language extraction and execution information. By comparing both to previously submitted defect reports, duplicates are easier to identify. By adding the additional step of execution information, it was found that duplicate defect reports were on average 20% easier to identify in the Firefox defect repository [96].

Using surface features, textual semantics, and graph clustering, Jalbert and Weimer were able to filter out 8% of duplicate defect reports from 29,000 defect reports obtained from the Mozilla project [97]. This approach was later improved by over 15% using discriminative models for information retrieval [98].

Sureka and Jalote proposed a solution using a character N-gram-based model. They found their solution to be effective on defect reports from the open-source Eclipse project [99].

Tan, Hu, and Chen proposed a keywords repository that can be added to when developers fix defects. That keyword repository can help in the detection of new duplicate defect reports [100].

Zhang and Lee proposed a defect rule based classification technique. They employed a developer feedback mechanism to improve the accuracy [101].

Davidson, Mohan, and Jensen found that, between small, medium, and large Free or Open Source Software projects, duplicate defect reports are more of a problem with medium-sized projects than with other sizes. This result is likely because the developers receive a large number of submissions, but lack the resources available for large projects [102].

D. Tools for Triage

Several research tools were developed for triage.

- **Bugzie**—For each defect report, Bugzie determines a ranked list of developers most likely capable of handling the issue. It solves this ranking problem by modeling the expertise of the developer toward a collection of technical terms extracted from the software artifacts of a project. It outperforms implementations using Naïve Bayes, SVM, and others [103].
- **DREX**—Developer Recommendation with k-nearest-neighbor search and Expertise ranking (DREX) suggests the assignment of developers to defect reports based on a K-Nearest-Neighbor search with defect similarity and expertise ranking. Using this method, developers are able to perform a recall of 60% on average [104].

VII. Conclusions, and Further Research

In this work, we reviewed 104 papers on the topic of defect reporting published since the NIST report through to 2012. We offer the following conclusions on the current state of the literature.

- Search Based Software Engineering (SBSE) is playing a greater role in the automatic fixing of defects by evolving code to meet the requirements for unit tests, or evolving the tests themselves.
- Identifying defects is becoming easier by limiting the search space to committed changes that have the greatest chance of introducing new defects. Work is being done in software inspection to scan for defects continuously during development.
- Algorithms are being developed to predict the attributes of defect reports. By predicting the severity, time to fix, and other factors, project managers have a consistent, better picture of the resource impact of new defects without waiting for triage.
- There is a disconnect between what is generally provided by the users in a defect report and what developers find useful. Organizations taking the time to closely evaluate what they are capturing in defect reports and trying to assist reporters in improving the quality of their report are paying off in a measurable way in terms of improved software quality.
- Triage of defect reports incurs a significant resource overhead. Automatic methods of assignment, categorization, and duplicate detection are being used with measured success. The advantages in this area are significant, with defect reports going to the developers that can resolve the issue the quickest, actual defect reports being separated from technical assistance and enhancement requests, and duplicates being aggregated into a master defect report for a singular issue.

We foresee many of these improvements being used in practice in the future. For practitioners, we comment on the notable tools developed to assist in improving the process of defect reporting as well as important facts learned through research.

- Tools such as GenProg, ReAssert, or AutoFix-E2 are available for developers to become familiar with the defect fixing benefits provided by evolutionary programming.
- Locating the associated code from a defect report is largely a manual process. Tools such as BugScout can assist developers in automatically locating that associated code from a defect report. We can expect both automatic defect detection and evolutionary computing to be a part of popular IDEs in the future.
- There is empirical evidence that many defect reports can be traced back to ambiguous requirements. Tools such as RETRO can assist in making that link to improve requirements and help identify domain-specific words that lead to defects.
- Developers expect a certain level of quality in a report; and when that quality is lacking, they use resources to hunt down issues, or at worst, miss recognizing important ones. A tool called CUEZILLA can help users improve the

quality of their report before submission. Other tools are becoming available to collect stack traces without invading privacy.

- Tools such as Bugzie can assist in the automatic assignment of expert developers to a defect. This method cuts down triage in a meaningful way by automatically linking a defect-prone component referenced in a defect report to source repository commits by a developer. In addition, duplicate defect reports are shown to cause more help than harm by providing additional context for defects. The duplicate reports should be aggregated in lieu of being closed.

There are several open research problems. In particular, we find these three general areas are needed to address the most broad standing issues in defect reporting.

- Automating defect detection and defect fixing requires more research into Search Based Software Engineering. Narrowing the search space and evolving code both improve this process, and should continue to be actively researched. Doing this work decreases the amount of defects that appear later in the product life-cycle, where issues arise in the quality of defect reports.
- The quality of defect reports suffer the most from a disconnect between what users report and what developers need. Ways to evolve defect reports from a description typed by a user toward a new taxonomy system or structured grammar for identifying defects could bridge this gap and prevent the ambiguity that developers often encounter when trying to understand a defect report.
- Metrics should improve to give project managers a better picture of the general risk associated with standing defect reports. Many metrics now take into account particular aspects of a defect; a metric that considers many of the predictable attributes of defect reports would give an overall better picture for project managers.

Despite more research being needed to achieve a largely automated defect process, great strides were made in achieving this goal. Most importantly for practitioners, research shows that scrutiny toward defect tracking systems pays off in a significant way for software quality. Finding simple tools for quality and triage of defect reports can repay the time invested with a measured increase in productivity.

References

[1] J. Li, T. Stâlhane, R. Conradi, and J. M. W. Kristiansen, "Enhancing defect tracking systems to facilitate software quality improvement," *IEEE Software*, vol. 29, pp. 59–66, Mar.-Apr. 2012.

[2] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. Evol. Comput., 2008. CEC 2008. (IEEE World Congress on Computational Intelligence)*, June 2008, pp. 162–168, IEEE Congress on. IEEE Computer Society.

[3] A. Arcuri, "On the automation of fixing software bugs," in *Proc. Companion 30th Int. Conf. Software Eng.*, 2008, ser. ICSE Companion '08, New York, NY, USA: ACM, pp. 1003–1006 [Online]. Available: http://doi.acm.org/10.1145/1370175.1370223

[4] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.* vol. 11, no. 4, pp. 3494–3514, June 2011 [Online]. Available: http://dx.doi.org/10.1016/j.asoc.2011.01.023

[5] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. 31st Int. Conf. Software Eng.*, 2009, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, pp. 364–374 [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070536

[6] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM* vol. 53, no. 5, pp. 109–116, May 2010 [Online]. Available: http://doi.acm.org/10.1145/1735223.1735249

[7] T. Nguyen, W. Weimer, C. Le Goues, and S. Forrest, "Using execution paths to evolve software patches," in *Pro. IEEE Int. Conf. Software Testing, Verification, Validation Workshops*, 2009, ser. ICSTW '09. Washington, DC, USA: IEEE Computer Society, pp. 152–153 [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2009.35

[8] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. 3rd Int. Conf. Software Testing, Verification, Validation (ICST)*, April 2010, pp. 65–74, IEEE Computer Society.

[9] D. White, A. Arcuri, and J. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, Aug. 2011.

[10] M. Harman, P. McMinn, J. de Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Eng., Verification, ser. Lecture NotesComput. Sci.*, B. Meyer and M. Nordio, Eds. : Springer Berlin/Heidelberg, 2012, vol. 7007, pp. 1–59, 10.1007/978-3-642-25231-0_1 [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25231-0_1

[11] M. Harman, "Software engineering meets evolutionary computation," *Computer*.

[12] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," in *Proc. 32nd ACM/IEEE Int. Conf. Software Eng.*, 2010, vol. 1, ser. ICSE '10. New York, NY, USA: ACM, pp. 255–264 [Online]. Available: http://doi.acm.org/10.1145/1806799.1806839

[13] J. L. Wilkerson and D. Tauritz, "Coevolutionary automated software correction," in *Proc. 12th Annu. Conf. Genet. Evol. Comput.*, 2010, ser. GECCO '10. New York, NY, USA: ACM, pp. 1391–1392 [Online]. Available: http://doi.acm.org/10.1145/1830483.1830739

[14] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proc. 19th Int. Symp. Software Testing Anal.*, 2010, ser. ISSTA '10. New York, NY, USA: ACM, pp. 61–72 [Online]. Available: http://doi.acm.org/10.1145/1831708.1831716

[15] Y. Pei, Y. Wei, C. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *Proc. 2011 26th IEEE/ACM Int. Conf.. Autom. Software Eng. (ASE),*, Nov. 2011, pp. 392–395, IEEE Computer Society.

[16] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "Reassert: Suggesting repairs for broken unit tests," in *Proc. ASE '09. 24th IEEE/ACM Int. Conf. Automated Software Eng.*, Nov. 2009, pp. 433–444, IEEE Computer Society.

[17] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *Proc. 19th Int. Symp. Software Testing Anal.*, 2010, ser. ISSTA '10. New York, NY, USA: ACM, pp. 207–218 [Online]. Available: http://doi.acm.org/10.1145/1831708.1831734

[18] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.* vol. 38, no. 1, pp. 54–72, Jan. 2012 [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.104

[19] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Software Eng.*, 2008, ser. ICSE '08. New York, NY, USA: ACM, pp. 181–190 [Online]. Available: http://doi.acm.org/10.1145/1368088.1368114

[20] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Software Eng.*, 2007, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, pp. 489–498 [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.66

[21] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Software Eng.*, 2009, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, pp. 78–88 [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070510

[22] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, May 2010, pp. 31–41, IEEE Computer Society.

[23] Y. Peng, G. Wang, and H. Wang, "User preferences based software defect detection algorithms selection using mcdm," *Inf. Sci.* vol. 191, pp. 3–13, May 2012 [Online]. Available: http://dx.doi.org/10.1016/j.ins.2010.04.019

[24] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Softw. Eng.* vol. 31, no. 6, pp. 466–480, Jun. 2005 [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.63

[25] J. DeMott, R. Enbody, and W. Punch, "Towards an automatic exploit pipeline," in *Proc. Int. Conf. Internet Technol. Secured Trans. (ICITST),* , Dec. 2011, pp. 323–329, IEEE Computer Society.

[26] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin, "Capture-recapture in software inspections after 10 years research-theory, evaluation and application," *J. Syst. Software* vol. 72, no. 2, pp. 249–264, 2004 [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121203000906

[27] C.-P. Chang, J.-L. Lv, and C.-P. Chu, "A defect estimation approach for sequential inspection using a modified capture-recapture model," in *Proc. 29th Annu. Int. Comput. Software Appl. Conf. (COMPSAC 2005)*, July 2005, vol. 1, pp. 41–46, IEEE Computer Society, Vol. 2.

[28] R. Bucholz and P. Laplante, "A dynamic capture-recapture model for software defect prediction," *Innovations Syst. Software Eng.* vol. 5, pp. 265–270, 2009, 10.1007/s11334-009-0099-y [Online]. Available: http://dx.doi.org/10.1007/s11334-009-0099-y

[29] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Software Eng.*, 2007, ser. PROMISE '07. Washington, DC, USA: IEEE Computer Society, p. 9 [Online]. Available: http://dx.doi.org/10.1109/PROMISE.2007.10

[30] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. 28th Int. Conf. Software Eng.*, 2006, ser. ICSE '06. New York, NY, USA: ACM, pp. 452–461 [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[31] N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, "Predicting software defects in varying development lifecycles using bayesian nets," *Inf. Softw. Technol.*.

[32] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Software Eng.*, 2010, ser. FSE '10. New York, NY, USA: ACM, pp. 97–106 [Online]. Available: http://doi.acm.org/10.1145/1882291.1882308

[33] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Software Eng. (ASE)*, Nov. 2011, pp. 263–272, IEEE Computer Society.

[34] K. S. Wasson, K. N. Schmid, R. R. Lutz, and J. C. Knight, "Using occurrence properties of defect report data to improve requirements," in *Proc. 13th IEEE Int. Conf. Requirements Eng.*, 2005, ser. RE '05. Washington, DC, USA: IEEE Computer Society, pp. 253–262 [Online]. Available: http://dx.doi.org/10.1109/RE.2005.77

[35] M. D'Ambros, M. Lanza, and M. Pinzger, ""a bug's life" visualizing a bug database," in *Proc. 4th IEEE Int.Workshop Visual. Software Understand. Anal.*, Jun. 2007, pp. 113–120, IEEE Computer Society.

[36] N. Ayewah and W. Pugh, "Learning from defect removals," in *Proc. 6th IEEE Int. Working Conf. Mining Software Repositorieson*, May 2009, pp. 179–182, IEEE Computer Society.

[37] S. Davies, M. Roper, and M. Wood, "A preliminary evaluation of text-based and dependency-based techniques for determining the origins of bugs," in *Proc. 18th Working Conf. Reverse Eng. (WCRE), 2011* , Oct. 2011, pp. 201–210, IEEE Computer Society.

[38] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: A case study of bug reports," in *Proc. 32nd ACM/IEEE Int. Conf. Software Eng.*, 2010, vol. 1, ser. ICSE '10. New York, NY, USA: ACM, pp. 505–514 [Online]. Available: http://doi.acm.org/10.1145/1806799.1806872

[39] P. Chilana, A. Ko, and J. Wobbrock, "Understanding expressions of unwanted behaviors in open bug reporting," in *Proc. IEEE Symp. Visual Lang. Human-Centric Comput. (VL/HCC)*, Sept. 2010, pp. 203–206, IEEE Computer Society.

[40] J. Ekanayake, J. Tappolet, H. C. Gall, and A. Bernstein, "Time variance and defect prediction in software projects," *Empirical Softw. Eng.* vol. 17, no. 4–5, pp. 348–389, Aug. 2012 [Online]. Available: http://dx.doi.org/10.1007/s10664-011-9180-x

[41] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?," in *Proc. 4th Int. Workshop Mining Software Repositories*, 2007, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, p. 1 [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13

[42] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proc. 2nd Int. Workshop Recommendation Syst. Software Eng.*, 2010, ser. RSSE '10. New York, NY, USA: ACM, pp. 52–56 [Online]. Available: http://doi.acm.org/10.1145/1808920.1808933

[43] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?," in *Proc. 8th Working Conf. Mining Software Repositories*, 2011, ser. MSR '11. New York, NY, USA: ACM, pp. 207–210 [Online]. Available: http://doi.acm.org/10.1145/1985441.1985472

[44] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in . *of the 7th Int. Conf. Predictive Models Software Engi.*, 2011, ser. Promise '11. New York, NY, USA: ACM, pp. 11:1–11:8 [Online]. Available: http://doi.acm.org/10.1145/2020390.2020401

[45] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," in *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, May 2010, pp. 1–10, IEEE Computer Society.

[46] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *Proc. 15th Eur. Conf. Software Maintenance Reengineering (CSMR)*, Mar. 2011, pp. 249–258, IEEE Computer Society.

[47] N. K. Nagwani and A. Bhansali, "A data mining model to predict software bug complexity using bug estimation and clustering," in *Proc. 2010 Int. Conf. Recent Trends Inf., Telecommun., Comput.*, 2010, ser. ITC '10. Washington, DC, USA: IEEE Computer Society, pp. 13–17 [Online]. Available: http://dx.doi.org/10.1109/ITC.2010.56

[48] A. Bacchelli, M. D'Ambros, and M. Lanza, "Are popular classes more defect prone?," in *Proc. 13th Int. Conf. Fundamental Approaches Software Eng.*, 2010, ser. FASE'10. Berlin, Heidelberg: Springer-Verlag, pp. 59–73 [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12029-9_5

[49] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows," in *Proc. 32nd ACM/IEEE Int. Conf. Software Eng.*, 2010, vol. 1, ser. ICSE '10. New York, NY, USA: ACM, pp. 495–504 [Online]. Available: http://doi.acm.org/10.1145/1806799.1806871

[50] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-I. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *Proc. 2010 17th Working Conf. Reverse Eng.*, 2010, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, pp. 249–258 [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.36

[51] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proc. 31st Int. Conf. Software Eng.*, 2009, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, pp. 298–308 [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070530

[52] A. Meneely, M. Corcoran, and L. Williams, "Improving developer activity metrics with issue tracking annotations," in *Proc. 2010 ICSE Workshop Emerging Trends Software Metrics*, 2010, ser. WETSoM '10. New York, NY, USA: ACM, pp. 75–80 [Online]. Available: http://doi.acm.org/10.1145/1809223.1809234

[53] D. Bertram, A. Voida, S. Greenberg, and R. Walker, "Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams," in *Proc. 2010 ACM Conf. Comput. Supported Cooperative Work*, 2010, ser. CSCW '10. New York, NY, USA: ACM, pp. 291–300 [Online]. Available: http://doi.acm.org/10.1145/1718918.1718972

[54] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proce. 4th Int. Workshop Mining Software Repositories*, 2007, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, p. 2 [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.7

[55] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information needs in bug reports: Improving cooperation between developers and users," in *Proc. 2010 ACM Conf. Comput. Supported Cooperative Work*, 2010, ser. CSCW '10. New York, NY, USA: ACM, pp. 301–310 [Online]. Available: http://doi.acm.org/10.1145/1718918.1718973

[56] A. J. Ko and P. K. Chilana, "How power users help and hinder open bug reporting," in *Proc. 28th Int. Conf. Human Factors Comput. Syst.*, 2010, ser. CHI '10. New York, NY, USA: ACM, pp. 1665–1674 [Online]. Available: http://doi.acm.org/10.1145/1753326.1753576

[57] S. Yadla, J. H. Hayes, and A. Dekhtyar, "Tracing requirements to defect reports: An application of information retrieval techniques," *Innovations Syst. Software Eng.* vol. 1, pp. 116–124, 2005, 10.1007/s11334-005-0011-3 [Online]. Available: http://dx.doi.org/10.1007/s11334-005-0011-3

[58] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Proc. IEEE Int. Conf. Software Maintenance*, Oct. 2008, pp. 346–355, IEEE Computer Society.

[59] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, "Recovering traceability links between source code and fixed bugs via patch analysis," in *Proc. 6th Int. Workshop Traceability Emerging Forms Software Eng.*, 2011, ser. TEFSE '11. New York, NY, USA: ACM, pp. 31–37 [Online]. Available: http://doi.acm.org/10.1145/1987856.1987863

[60] N. Bettenburg, S. Just, A. Schröter, C. Weiß, R. Premraj, and T. Zimmermann, "Quality of bug reports in eclipse," in *Proc. 2007 OOPSLA Workshop Eclipse Technol. EXchange*, 2007, ser. eclipse '07. New York, NY, USA: ACM, pp. 21–25 [Online]. Available: http://doi.acm.org/10.1145/1328279.1328284

[61] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, Frequently asked questions in bug reports Department of Computer Science. University of Calgary, Tech. Rep. 2009-924-03, March 2009.

[62] E. I. Laukkanen and M. V. Mantyla, "Survey reproduction of defect reporting in industrial software development," in *Proc. 2011 Int. Symp. Empirical Software Eng. Measur.*, 2011, ser. ESEM '11. Washington, DC, USA: IEEE Computer Society, pp. 197–206 [Online]. Available: http://dx.doi.org/10.1109/ESEM.2011.28

[63] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in *Proc. IEEE Symp. Visual Lang. Human-Centric Computi.*, 2008, ser. VLHCC '08. Washington, DC, USA: IEEE Computer Society, pp. 82–85 [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2008.4639063

[64] A. J. Ko, B. A. Myers, and D. H. Chau, "A linguistic analysis of how people describe software problems," in *Proc. Vis. Lang. Human-Centric Comput.*, 2006, ser. VLHCC '06. Washington, DC, USA: IEEE Computer Society, pp. 127–134 [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2006.3

[65] W. Weimer, "Patches as better bug reports," in *Proc. 5th Int. Conf. Generative Program. Component Eng.*, 2006, ser. GPCE '06. New York, NY, USA: ACM, pp. 181–190 [Online]. Available: http://doi.acm.org/10.1145/1173706.1173734

[66] M. Castro, M. Costa, and J.-P. Martin, "Better bug reporting with better privacy," *SIGARCH Comput. Archit. News* vol. 36, no. 1, pp. 319–328, March 2008 [Online]. Available: http://doi.acm.org/10.1145/1353534.1346322

[67] B. Dit and A. Marcus, "Improving the readability of defect reports," in *Proc. 2008 In. Workshop Recommendation Syst. Software Eng.*, 2008, ser. RSSE '08. New York, NY, USA: ACM, pp. 47–49 [Online]. Available: http://doi.acm.org/10.1145/1454247.1454265

[68] B. Dit, "Measuring the semantic similarity of comments in bug reports," in *Proc. 1st Int. ICPC2008 Workshop Semantic Technol. Syst. Maintenance*, 2008, ser. STSM '08. IEEE Computer Society [Online]. Available: http://www.cs.wm.edu/~denys/pubs/Dit-STSM08.pdf

[69] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *Software Eng.—Companion*, May 2009, vol. 2009, ICSE-Companion 2009. 31st International Conference on. IEEE Computer Society, pp. 247–250.

[70] A. Schroter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?," in *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, May 2010, pp. 118–121, IEEE Computer Society.

[71] T. Nguyen, B. Adams, and A. Hassan, "A case study of bias in bug-fix datasets," in *Proc. 2010 17th Working Conf. Reverse Eng. (WCRE),*, Oct. 2010, pp. 259–268, IEEE Computer Society.

[72] J. Sun, "Why are bug reports invalid?," in *Proc. 2011 4th IEEE Int. Conf. Software Testing, Verification Validation*, 2011, ser. ICST '11. Washington, DC, USA: IEEE Computer Society, pp. 407–410 [Online]. Available: http://dx.doi.org/10.1109/ICST.2011.43

[73] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Software Eng.*, 2008, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, pp. 308–318 [Online]. Available: http://doi.acm.org/10.1145/1453101.1453146

[74] S. Herbold, J. Grabowski, S. Waack, and U. Bünting, "Improved bug reporting and reproduction through non-intrusive gui usage monitoring and automated replaying," in *Proc. 2011 IEEE 4th Int. Conf. Software Testing, Verification Validation Workshops*, 2011, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, pp. 232–241 [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.66

[75] P. A. Laplante and N. Ahmad, "Pavlov's bugs: Matching repair policies with rewards," IT Professional.

[76] D. Čubranić, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Software Eng. Knowledge Eng.*. : KSI Press, 2004, pp. 92–97.

[77] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proc. 28th Int. Conf. Software Eng.*, 2006, ser. ICSE '06. New York, NY, USA: ACM, pp. 361–370 [Online]. Available: http://doi.acm.org/10.1145/1134285.1134336

[78] J. Anvik, "Automating bug report assignment," in *Proc. 28th Int. Conf. Software Eng.*, 2006, ser. ICSE '06. New York, NY, USA: ACM, pp. 937–940 [Online]. Available: http://doi.acm.org/10.1145/1134285.1134457

[79] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.* vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011 [Online]. Available: http://doi.acm.org/10.1145/2000791.2000794

[80] W. Zou, Y. Hu, J. Xuan, and H. Jiang, "Towards training set reduction for bug triage," in *Proc. IEEE 35th Annu. Comput. Software Appl. Conf. (COMPSAC), 2011* , July 2011, pp. 576–581, IEEE Computer Society.

[81] P. Bhattacharya, I. Neamtiu, and C. R. Shelton, "Automated, highly-accurate, bug assignment using machine learning and tossing graphs," *J. Syst. Softw.* vol. 85, no. 10, pp. 2275–2292, Oct. 2012 [Online]. Available: http://dx.doi.org/10.1016/j.jss.2012.04.053

[82] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meeting Eur. Software Eng. Conf. ACM SIGSOFT Symp. Found. Software Eng.*, 2009, ser. ESEC/FSE '09. New York, NY, USA: ACM, pp. 111–120 [Online]. Available: http://doi.acm.org/10.1145/1595696.1595715

[83] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang, "An empirical study on bug assignment automation using chinese bug data," in *Proc. 2009 3rd Int. Symp. Empirical Software Eng. Measur.*, 2009, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, pp. 451–455 [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5315994

[84] D. Matter, "Who Knows About That Bug?—Automatic Bug Report Assignment With a Vocabulary-Based Developer Expertise Model" Ph.D. dissertation, University of Berne, Berne, Switzerland, 2009 [Online]. Available: http://scg.unibe.ch/archive/masters/Matt09b.pdf

[85] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *Proc. 2009 6th IEEE Int. Working Conf. Mining Software Repositories*, 2009, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, pp. 131–140 [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069491

[86] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Hammad, "Assigning change requests to software developers," *J. Software: Evol. Process* vol. 24, no. 1, pp. 3–33, 2012 [Online]. Available: http://dx.doi.org/10.1002/smr.530

[87] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Software Eng.*, 2003, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, pp. 465–475 [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776872

[88] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proc. 2008 Conf. Center for Adv. Studies Collaborative Res.: Meeting Minds*, 2008, ser. CASCON '08. New York, NY, USA: ACM, pp. 23:304–23:318 [Online]. Available: http://doi.acm.org/10.1145/1463788.1463819

[89] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *Proc. 7th IEEE Working Conf. Mining Software Repositories (MSR)*, May 2010, pp. 11–20, IEEE Computer Society.

[90] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic bug triage using semi-supervised text classification," SEKE Knowledge Systems Institute Graduate School, 2010, pp. 209–214.

[91] J. Xiao and W. Afzal, "Search-based resource scheduling for bug fixing tasks," in *Proc. 2nd Int. Symp. Search Based Software Eng.*, 2010, ser. SSBSE '10. Washington, DC, USA: IEEE Computer Society, pp. 133–142 [Online]. Available: http://dx.doi.org/10.1109/SSBSE.2010.24

[92] F. Khomh, B. Chan, Y. Zou, and A. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *Proc. 18th Working Conf. Reverse Eng. (WCRE)*, Oct. 2011, pp. 261–270, IEEE Computer Society.

[93] Y. Cavalcanti, P. da Mota Silveira Neto, E. de Almeida, D. Lucrédio, C. da Cunha, and S. de Lemos Meira, "One step more to understand the bug report duplication problem," in *Proc. Brazilian Symp. Software Eng. (SBES)*, Oct. 2010, pp. 148–157, IEEE Computer Society.

[94] Y. Cavalcanti, E. de Almeida, C. da Cunha, D. Lucré anddio, and S. de Lemos Meira, "An initial study on the bug report duplication problem," in *Proc. 14th Eur. Conf. Software Maintenance Reeng. (CSMR)*, March 2010, pp. 264–267, IEEE Computer Society.

[95] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful . . . really?," in *Proc. IEEE Int. Conf. Software Maintenance*, Oct. 2008, pp. 337–345, IEEE Computer Society.

[96] 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 *Proc. 30th Int. Conf. Software Eng.*, 2008, ser. ICSE '08. New York, NY, USA: ACM, pp. 461–470 [Online]. Available: http://doi.acm.org/10.1145/1368088.1368151

[97] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS DCC*, June 2008, pp. 52–61, IEEE Computer Society.

[98] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32nd ACM/IEEE Int. Conf. Software Eng.*, 2010, vol. 1, ser. ICSE '10. New York, NY, USA: ACM, pp. 45–54 [Online]. Available: http://doi.acm.org/10.1145/1806799.1806811

[99] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proc. 2010 Asia Pacific Software Eng. Conf.*, 2010, ser. APSEC '10. Washington, DC, USA: IEEE Computer Society, pp. 366–374 [Online]. Available: http://dx.doi.org/10.1109/APSEC.2010.49

[100] S. Tan, S. Hu, and L. Chen, "A framework of bug reporting system based on keywords extraction and auction algorithm," in *Proc. 5th Annu. ChinaGrid Conf.*, 2010, ser. CHINAGRID '10. Washington, DC, USA: IEEE Computer Society, pp. 281–284 [Online]. Available: http://dx.doi.org/10.1109/ChinaGrid.2010.13

[101] T. Zhang and B. Lee, "A bug rule based technique with feedback for classifying bug reports," in *Proc. 2011 IEEE 11th Int. Conf. Comput. Inf. Technol.*, 2011, ser. CIT '11. Washington, DC, USA: IEEE Computer Society, pp. 336–343 [Online]. Available: http://dx.doi.org/10.1109/CIT.2011.90

[102] J. Davidson, N. Mohan, and C. Jensen, "Coping with duplicate bug reports in free/open source software projects," in *Proc. IEEE Symp. Visual Lang. Human-Centric Comput. (VL/HCC)*, Sep. 2011, pp. 101–108, IEEE Computer Society.

[103] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Software Eng.*, 2011, ser. ESEC/FSE '11. New York, NY, USA: ACM, pp. 365–375 [Online]. Available: http://doi.acm.org/10.1145/2025113.2025163

[104] W. Wu, W. Zhang, Y. Yang, and Q. Wang, "Drex: Developer recommendation with k-nearest-neighbor search and expertise ranking," in *Proc. 2011 18th Asia Pacific Software Eng. Conf. (APSEC)*, Dec. 2011, pp. 389–396, IEEE Computer Society.

Jonathan D. Strate (S'10–M'11) received the B.S. degree in computer science from The Pennsylvania State University, State College, and is working toward the M.Eng degree in software engineering at The Pennsylvania State University, Malvern.

He is currently a Software Engineer at Analytical Graphics, Inc. in Exton, PA where he is responsible for the user interface for an upcoming product in Space Situational Awareness. He additionally performs consulting work for commercial web site design and development. His research interests include the software development process, project management, and human-computer interaction.

Phillip A. Laplante (M'86–SM'90–F'08) received the B.S. degree in systems planning and management, M.Eng. degree in electrical engineering, and the Ph.D. degree in computer science from the Stevens Institute of Technology, Hoboken, NJ, in 1983, 1986, and 1990, respectively, and the M.B.A. degree from the University of Colorado, Colorado Springs, in 1999.

He is a Professor of software engineering with Pennsylvania State University, Malvern, and is currently serving on the Reliability Society Adcom.