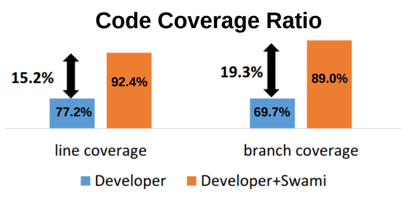
Due to the ubiquity of software and software rarely being free of bugs, the impact of software failures has never been greater. In 2017, 606 major software failures caused 314 companies to lose $1.7 trillion and affected half the world population [10]. Software engineers spent 35–50% of their time validating and debugging software, and the cost of debugging, testing, and verification is estimated to account for 50-75% of the total budget of software development, amounting to more than $100 billion annually [14]. Software can fail because of wrong or conflicting software requirements, bad architecture design, buggy source code, and inadequate testing. Software debugging involves detecting and fixing all such causes. *The overarching goal of my research is to automate software debugging by using natural language software artifacts and aid software engineers in developing high-quality software.* I have published my research in top software engineering conferences and journals such as TSE [1, 22], ICSE [17, 18, 21], FSE [7], EMSE [20], and ASE [6].

1 Using Natural Language Software Artifacts to Automate Software Debugging

The software development process includes eliciting requirements, creating software design, implementing source code, testing, and maintenance. Each of these activities uses and generates documents, many of which contain natural language descriptions, specifications, and constraints described at various levels of abstraction that range from high-level specifications such as functional requirements described in software requirements specification documents to low-level specifications such as bug reports and code comments describing implemented code. An incomplete understanding of these scattered software artifacts causes software engineers to introduce defects while performing the software development activities and makes debugging hard. To support software debugging, I have developed automated techniques that use the natural language software artifacts to aid software engineers in: (1) synthesizing tests with oracles (Section 1.1), (2) fixing bugs in the source code (Section 1.2), and (3) eliciting correct requirements (Section 1.3).

1.1 Synthesizing Tests with Oracles Using Structured Natural Language Specifications

Software testing involves examining the behavior of software to discover potential bugs. Given an input for software, the challenge of distinguishing the desired, correct behavior from potentially incorrect behavior is called the test oracle problem. Existing automated test generation tools (e.g., EvoSuite, JDoctor) either use the source code of the software or specific annotations from code comments (e.g., JavaDoc) to compute test oracles. Computing test oracles from source code assumes that the software works correctly. While this assumption is acceptable for defect benchmarks that provide both incorrect and correct versions of software code for defect, it is not acceptable in practical scenarios where only one version of software code exists. The approaches that use code comments can be useful in practice; however, code comments are often not updated by developers while updating code, and this inconsistency leads to generating incorrect tests. To address these limitations, I developed Swami [18], a technique that extracts test oracles from structured natural language software specifications and generates executable tests. Swami focuses on exceptional behavior and boundary conditions that often cause software failures and for which developers often fail to write tests [8]. I used Swami to generate tests from the official JavaScript specification (ECMA-262) and used the generated tests to evaluate two popular JavaScript implementations, Mozilla Rhino and Node.js. I found that 98.4% of the generated tests were precise to the ECMA-262 specification. Using Swami tests, I identified a previously unknown bug[1], missing features[2], and semantic ambiguities in the ECMA-262 specification. Further, Swami generated tests for behavior uncovered by developer-written tests for 12 Rhino methods. The average statement coverage for these methods improved by 15.2%, and the average branch coverage improved by 19.3%. Thus, using Swami enables developers to automatically improve



[1]`https://github.com/mozilla/rhino/issues/522`
[2]`https://github.com/mozilla/rhino/issues/521`

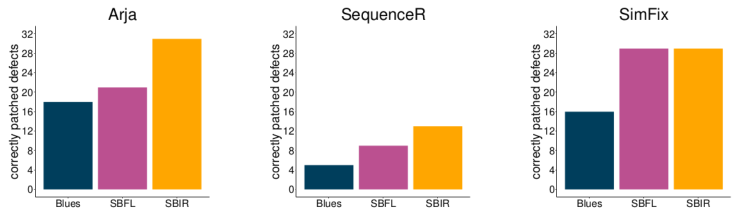
their tests without manually going through the ~500-page ECMA-262 specification document to identify and create tests for the exceptional and boundary conditions.

1.2 Fixing Bugs in Software Using Both Tests and Bug Reports

Automated Program Repair is an active research area that aims to develop tools to reduce the cost of manually fixing bugs by automatically producing patches [5, 12]. While repair tools are successful in patching real-world defects, often the patches repair some functionality encoded by the tests, while simultaneously breaking other undertested functionality. Thus, *quality* or correctness of the produced patches is a critical concern. I found that the *patch overfitting* problem — patches pass a particular set of test cases supplied to the repair tool but fail to generalize to the desired specification — is common as more than 50% of the patches produced by repair tools overfit to the tests used in the repair process [22]. This makes developers lose trust in the repair tools deterring their wide-scale adoption in practice [2].

To address the patch overfitting problem, my colleagues and I developed SOSRepair [1], a novel semantics-search-based repair technique that produces patches by replacing likely buggy regions of software code with semantically-similar human-written code fragments, and does so at a higher level of granularity (replacing 3–7 lines of code) than prior approaches, which replaced a single line at a time. SOSRepair builds on the ideas from SearchRepair [9], an existing repair tool that demonstrated that higher granularity, semantic-based changes can improve patch quality. However, SearchRepair could only work on small C programs with fewer than 25 lines. To make SOSRepair apply, at scale, to real-world defects, we redesigned the entire approach and developed a conceptually novel method to perform semantic code search. SOSRepair patches defects in large, real-world, multi-million-line C projects. On a subset of 65 defects from the ManyBugs benchmark [11], SOSRepair patched 22 (34%) defects that included defects which previous state-of-the-art repair techniques did not patch. On these 22 defects, SOSRepair patched more defects correctly (9, 41%) than the prior techniques. Analyzing the cause for defects that SOSRepair did not patch correctly, we found the fault localization accuracy was a key factor in SOSRepair's success. Manually improving fault localization allowed SOSRepair to patch 23 (35%) defects, of which 16 (70%) were correctly patched. These findings motivated me to develop a more accurate fault localization technique for program repair as described next.

Most repair tools use test suites or bug reports to identify the potentially buggy program elements (e.g., statements, methods, or classes). I developed SBIR [19], the first unsupervised technique that combines information from bug reports and test executions to identify the buggy program elements. Unlike existing supervised learning-based techniques that combine multiple fault localization techniques but are ill-suited for repair tools, SBIR is easy to integrate into repair tools. On a dataset of 815 real-world bugs, SBIR consistently outperformed test-suite-based (SBFL) and bug-report-based (Blues) fault localization techniques. For example, SBIR identified a buggy program statement as the most suspicious for 18.3% of the bugs, while SBFL identified for 11.0% and Blues identified for 4.0%. SBIR even outperformed combibing results of SBFL and Blues. Next, I evaluated three state-of-the-art repair tools, Arja, SequenceR, and SimFix, on 689 real-world bugs and compared the repairs produced when using SBIR, SBFL, and Blues. Arja and SequenceR significantly benefited from SBIR: Arja using SBIR repaired 47.6% more bugs than using SBFL and 72.2% more bugs than using Blues; for SequenceR, it's 44.4% and 160% more bugs, respectively. SimFix, which employs tricks to overcome poor fault localization, repaired the same number of bugs using SBIR as when using SBFL, and 81.3% more bugs than using Blues. This is the first attempt at using both test suites and bug reports for program repair and, the findings show that repair tools benefit greatly, suggesting a fruitful direction for research into using more natural-language software artifacts to improve automated program repair.



1.3 Using Natural Language Specifications to Elicit Software Requirements

In typical distributed software development scenarios, requirement documents are written in consultation with the clients and are handed over to offshore teams of developers. The documents need to be studied and analyzed before the implementation can begin. Large IT companies generate an enormous amount of documented knowledge over the years. Project managers always hope to reuse this knowledge for new projects, but engineers are swamped by the task of having to read the available documents to identify critical pieces of knowledge. Thus, companies rely heavily on the expertise of requirements engineers to elicit correct software requirements based on their experience and knowledge gained by working on past projects. However, this manual process is error-prone as engineers tend to miss out eliciting all software requirements and sometimes select conflicting software requirements that lead to developing incorrect software and costs the companies in hefty fines. To support eliciting correct and complete software requirements, I developed a method and tool [6] to automatically extract system use-cases and validations from the natural language software requirements documents of the previously developed software systems. I integrated my method in the knowledge-assisted requirements configurator [4], a tool that uses an ontology-based model to capture software requirements of past projects and can be used to configure and elicit requirements for new software. The configurator recommended engineers elicit complementary software requirements (based on the software requirements selected) and warned them when engineers chose conflicting requirements. Evaluating the configurator to elicit requirements for developing six real-world software systems for various insurance companies showed the configurator enabled requirements engineers to elicit 92.2% of the requirements, on average. While using the traditional process they elicited only 48.1% of the requirements, on average [3]. Thus, using the configurator significantly boosted engineer's productivity and lowered the risk of missing out on critical requirements or selecting conflicting ones that may cause software failures and incur a loss to the company.

2 Research Vision: Automated, End-To-End Software Debugging

My overarching research goal is to automate the end-to-end software debugging process by developing tools that can automatically synthesize or repair software requirements, design, source code, and tests from natural language software specifications. This will allow software engineers to focus on creative tasks of developing new features and reduce the cost of software debugging.

Developing automated tools to support software architects. I have built tools to aid software engineers to synthesize tests, fix bugs, and elicit requirements. My next goal is to develop tools that help engineers to correctly design software based on the software requirements. Software requirements often lack the details needed to make informed architectural decisions. For example, a functional requirement for an e-commerce system that states that *system should notify the status of the purchase order after the customer makes the payment* does not provide details about how (e.g., email or text) the notification should be sent, what customer contact details are known, and how to notify when a customer purchases as a guest as opposed to a member. The software design documents clarify such details by describing what the software should do exactly. Accordingly, software engineers select the best suitable architecture for development. Software architects manually prepare detailed system use-cases along with pre-conditions and assumptions, appropriate conceptual and system architecture design to implement the use-cases, system interfaces (e.g., user interfaces, data interfaces, and communication interfaces), and low-level details of classes and methods. Software developers closely follow the design documents when implementing the source code. Thus, any flaw in the design documents can lead to developing an incorrect software. While there exists work on assisting software architects in manually writing correct design documents and automatically detecting architectural flaws from source code, I will develop techniques that extract the knowledge from correctly written design documents to: (1) recommend right software architecture designs (e.g., building on my previous work on eliciting correct software requirements (Section 1.3)) and (2) automatically fix architectural flaws by refactoring the source code. For (2), assuming a correct design document, I will first develop techniques to identify if the implemented code is consistent with the design document. If not, my technique will identify a minimal set of software code changes to make the code consistent with the design. Finally, I will develop techniques for automated code refactoring.

Extending repair tools to fix hard and important defects. Before attempting to repair defects, developers often ask questions such as "How difficult will this defect be to fix?" and "Is the defect worth fixing?". Thus, to make program repair tools useful for practitioners, researchers need to know what kind of defects developers find worth fixing and are hard to fix. To that end, I developed a framework [20] to objectively characterize defects that are hard and important from developers' perspective and used the framework to evaluate repair tools. I found that modern repair tools fix defects that are simple yet important for developers. Researchers [13, 16] have built on my work to produce more evaluation metrics and frameworks that aim to boost the development of practical and reliable program repair tools. I will work on extending the capability of repair techniques to fix defects that are hard and important for developers. I will address this using two different approaches. First, the top-down approach, for which I will develop techniques to independently improve the three main steps (localizing bugs, effectively generating candidate patches, and validating candidate patches for correctness) of the repair process. For this, I will extend my work on improving fault localization [19] and test generation [18] by using the recent advancements made in the area of natural language processing that uses deep leaning-based techniques [15]. Second, the bottom-up approach, for which I will develop new repair algorithms that can input the expected software behavior and the bug information available in different software artifacts (e.g., bug reports, failing tests, stack trace, requirements documents, etc.), which are not used by modern repair tools to repair bugs. Recent advancements[3] in using deep learning-based techniques to generate tests and repair bugs have shown promising results. I will device such deep learning-based repair techniques that can learn from different information sources and produce correct repairs.

Developing software debugging tools for machine learning-based software. Practitioners are increasingly shifting to develop software using modern machine learning and deep learning techniques. As the software using these new techniques are fundamentally different from traditional ones, existing software debugging techniques do not directly apply to such software. For example, fixing defects in an ML-based software requires inspecting both source code and the dataset used for training the software. Methods to automatically test, detect and fix bugs in ML-based software are active research areas. I will develop techniques that can localize and repair defects in ML-based software. This will require developing novel techniques that can localize and fix bugs in datasets (e.g., detecting if a bug is caused because the dataset is skewed with respect to some features) and source code that typically builds on complex ML frameworks such as PyTorch and TensorFlow. Further, new research challenges have emerged by using new programming paradigms. For example, ensuring ML-based software is fair to its users has become an important research problem because of its increasing societal impact. I will develop novel debugging techniques to address such new challenges.

References

[1] Afsoon Afzal, **Manish Motwani**, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. SOSRepair: Expressive] Semantic Search for Real-World Program Repair. *IEEE Transactions on Software Engineering (TSE)*, 2021.

[2] Gene M Alarcon, Charles Walter, Anthony M Gibson, Rose F Gamble, August Capiola, Sarah A Jessup,

[3] `https://ml4code.github.io/`

and Tyler J Ryan. Would you fix this code for me? effects of repair source and commenting on trust in code repair. *Systems*, 8(1):8, 2020.

[3] Preethu Rose Anish and Smita Ghaisas. Product knowledge configurator for requirements gap analysis and customizations. In *IEEE International Requirements Engineering Conference (RE)*, pages 437–443, 2014.

[4] Preethu Rose Anish, Shashi Kant Sharma, **Manish Motwani**, and Smita Ghaisas. Knowledge-assisted product requirements configurator. In *2013 International Workshop on Product LinE Approaches in Software Engineering (PLEASE)*, pages 29–32, 2013.

[5] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Trans. on Software Eng.*, 45(01):34–67, 2019.

[6] Smita Ghaisas, **Manish Motwani**, and Preethu Rose Anish. Detecting system use cases and validations from documents. In *2013 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 568–573, 2013.

[7] Smita Ghaisas, **Manish Motwani**, Balaji Balasubramaniam, Anjali Gajendragadkar, Rahul Kelkar, and Harrick Vin. Towards automating the security compliance value chain. In *ACM Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, page 1014–1017, New York, NY, USA, 2015.

[8] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 213–224, Saarbrücken, Germany, July 2016.

[9] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE)*, pages 295–306, Lincoln, NE, USA, November 2015.

[10] Herb Krasner. The cost of poor software quality in the us: A 2018 report. In *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2018.

[11] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The manybugs and introclass benchmarks for automated repair of C programs. *IEEE TSE*, 41(12):1236–1256, December 2015.

[12] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM (CACM)*, 62(12):56–65, November 2019.

[13] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2020.

[14] Devon H O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, 2017.

[15] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. A survey on machine learning techniques for source code analysis. *arXiv preprint arXiv:2110.09610*, 2021.

[16] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140, 2018.

[17] **Manish Motwani**. High-quality automated program repair. In *IEEE/ACM International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 309–314, 2021.

[18] **Manish Motwani** and Yuriy Brun. Automatically generating precise oracles from structured natural language specifications. In *IEEE International Conference on Software Engineering (ICSE)*, pages 188–199, 2019.

[19] **Manish Motwani** and Yuriy Brun. Automatically repairing programs using both tests and bug reports. *arXiv:2011.08340*, 2020.

[20] **Manish Motwani**, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering (EMSE)*, 23(5):2901–2947, 2018.

[21] **Manish Motwani**, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. [journal first] do automated program repair techniques repair hard and important bugs? In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 25–25, 2018.

[22] **Manish Motwani**, Mauricio Soto, Yuriy Brun, Rene Just, and Claire Le Goues. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering*, 2021.