

Automated Program Repair

Edited by

Sunghun Kim¹, Claire Le Goues², Michael Pradel³, and
Abhik Roychoudhury⁴

1 HKUST – Kowloon, HK, hunkim@cse.ust.hk

2 Carnegie Mellon University – Pittsburgh, US, clegoues@cs.cmu.edu

3 TU Darmstadt, DE, michael@binaervarianz.de

4 National University of Singapore, SG, abhik@comp.nus.edu.sg

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 17022 “Automated Program Repair”. The seminar participants presented and discussed their research through formal and informal presentations. In particular, the seminar covered work related to search-based program repair, semantic program repair, and repair of non-functional properties. As a result of the seminar, several participants plan to launch various follow-up activities, such as a program repair competition, which would help to further establish and guide this young field of research.

Seminar January 8–13, 2017 – <http://www.dagstuhl.de/17022>

1998 ACM Subject Classification D.2 Software Engineering, D.2.5 [Software Engineering] Testing and Debugging

Keywords and phrases Program repair, program analysis, software engineering

Digital Object Identifier 10.4230/DagRep.7.1.19


1 Executive Summary

Sunghun Kim

Claire Le Goues

Michael Pradel

Abhik Roychoudhury

License  Creative Commons BY 3.0 Unported license
© Sunghun Kim, Claire Le Goues, Michael Pradel, and Abhik Roychoudhury

Software engineering targets the creation of software for myriad platforms, deployed over the internet, the cloud, mobile devices and conventional desktops. Software now controls cyber-physical systems, industrial control systems, and “Internet of Things” devices, and is directly responsible for humanity’s economic well-being and safety in numerous contexts. It is therefore especially important that engineers are able to easily write error-free software, and to quickly find and correct errors that do appear. Future generation programming environments must not only employ sophisticated strategies for localizing software errors, but also strategies for automatically patching them.

Recent years have seen an explosive growth in research on automated program repair, with proposed techniques ranging from pure stochastic search to pure semantic analysis. The Dagstuhl Seminar in January 2017 studies the problem of automated repair in a holistic fashion. This will involve a review of foundational techniques supporting program repair, perspectives on current challenges and future techniques, and emerging applications. The aim



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Automated Program Repair, *Dagstuhl Reports*, Vol. 7, Issue 1, pp. 19–31

Editors: Sunghun Kim, Claire Le Goues, Michael Pradel, and Abhik Roychoudhury



DAGSTUHL
REPORTS

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is to broadly discuss and revisit underlying assumptions and methods towards the integration of automated patch synthesis into futuristic programming environments.

Conceptually, applications of program repair step far beyond the general goal of ensuring software quality, and the subject is relevant to a broad range of research areas. It is of obvious importance in software testing and analysis, because repair goes hand in hand with traditional testing and debugging activities. It is relevant to researchers in programming languages and systems, e.g., to study language and system-level techniques that integrate patch suggestions during development. The topic is relevant to researchers in systems security, as repair approaches may be customizable to patching vulnerabilities in both application and systems software. Researchers in formal methods may provide insight for provably correct repair, given appropriate correctness specifications. Finally, the topic is connected to human computer interaction in software engineering, since the repair process, if not fully automated, may involve eliciting appropriate developer feedback and acting on it accordingly.

At a technical level, one of the key discussion topics has been the correctness specifications driving the repair process. Most previous work in this domain has relied on test suites as partial correctness specifications. While often available, test suites are typically inadequate for fully assessing patch correctness. Alternative quality specifications, such as “minimality”, could be explored. In addition, heavier-weight specifications, such as assertions, may provide stronger functional guarantees, leaving open the research challenge both in how to use them and how they may be derived to guide a repair process. Given the appropriate correctness specification, the task of repair usually involves three steps: localizing an error to a small set of potentially-faulty locations, deriving values/constraints for the computation desired at the faulty locations, and constructing “fix” expressions/statements that satisfy these values/constraints. Each of the three steps can be accomplished by a variety of methods, including heuristic search, symbolic analysis and/or constraint solving techniques. This allows for an interesting interplay for an entire design space of repair techniques involving ingenious combinations of search-based techniques and semantic analysis being employed at the different steps of the repair process.

The Dagstuhl Seminar has attracted researchers and practitioners from all over the world, comprising participants active in the fields of software engineering, programming languages, machine learning, formal methods, and security. As a result of the seminar, several participants plan to launch various follow-up activities, such as a program repair competition, which would help to further establish and guide this young field of research, and a journal article that summarizes the state of the art in automated program repair.

2 Table of Contents

Executive Summary

Sunghun Kim, Claire Le Goues, Michael Pradel, and Abhik Roychoudhury 19

Overview of Talks

Quality and applicability of automated repair <i>Yuriy Brun</i>	23
Automatic Tradeoffs: Accuracy and Energy <i>Jonathan Dorn</i>	23
Deep Learning for Program Repair <i>Aditya Kanade</i>	23
Prex: Finding Guidance for Forward and Backward Porting of Linux Device Drivers <i>Julia Lawall</i>	24
Automated Inference of Code Transforms and Search Spaces for Patch Generation <i>Fan Long</i>	24
Learning-based Program Repair <i>Fan Long</i>	24
ASTOR: A Program Repair Library for Java <i>Matías Sebastián Martínez</i>	25
Combining syntactic and semantic repair <i>Sergey Mehtaev</i>	25
Antifragile Software and Correctness Attraction <i>Martin Monperrus</i>	26
Detecting and repairing performance bugs that have non-intrusive fixes <i>Adrian Nistor</i>	26
Automated Software Transplantation <i>Justyna Petke</i>	26
Understanding and Automatically Preventing Injection Attacks on Node.js <i>Michael Pradel</i>	27
Anti-patterns in Search-Based Program Repair <i>Mukul Prasad</i>	27
Semantic Techniques for Program Repair <i>Abhik Roychoudhury</i>	28
Automated techniques for fixing performance issues in JavaScript applications <i>Marija Selakovic</i>	28
I Get by With a Little Help From My Friends: Crowdsourcing Program Repair <i>Kathryn T. Stolee</i>	28
Towards Trustworthy Program Repair <i>Yingfei Xiong</i>	29
How Developers Diagnose and Repair Software Bugs (and what we can do about it) <i>Andreas Zeller</i>	29

22 17022 – Automated Program Repair

Automated Test Reuse via Code Transplantation <i>Tianyi Zhang</i>	29
Participants	31

3 Overview of Talks

3.1 Quality and applicability of automated repair

Yuriy Brun (University of Massachusetts – Amherst, US)

License © Creative Commons BY 3.0 Unported license
© Yuriy Brun

Program repair offers great promise for reducing manual effort involved in software engineering, but only if it can produce high-quality patches for defects that are important and hard for humans to fix manually. This talk presents an objective measure of repair quality, identifies shortcomings in existing automated repair techniques in terms of the quality of the patches they produce, and tackles the problem of identifying if program repair techniques can repair important and hard defects.

3.2 Automatic Tradeoffs: Accuracy and Energy

Jonathan Dorn (University of Virginia – Charlottesville, US)

License © Creative Commons BY 3.0 Unported license
© Jonathan Dorn

Tradeoffs between competing objectives are an important part of software development and usability. However, the development effort to implement different tradeoffs is time consuming and potentially error-prone. In this work, we balance competing functional and non-functional properties via automatic program transformations. We apply multi-objective search to simultaneously optimize energy consumption and output accuracy of assembly programs. We find that relaxing the accuracy requirements enables greater energy reductions within the constraint of human-acceptability. We also find that our search-based approach identifies better tradeoff opportunities than less general techniques like loop perforation.

3.3 Deep Learning for Program Repair

Aditya Kanade (Indian Institute of Science – Bangalore, IN)


License © Creative Commons BY 3.0 Unported license
© Aditya Kanade

The problem of automatically fixing programming errors is a very active research topic in software engineering. This is a challenging problem as fixing even a single error may require analysis of the entire program. In practice, a number of errors arise due to programmer's inexperience with the programming language or lack of attention to detail. We call these common programming errors. These are analogous to grammatical errors in natural languages.

Compilers detect such errors, but their error messages are usually inaccurate. In this work, we present an end-to-end solution, called DeepFix, that can fix multiple such errors in a program without relying on any external tool to locate or fix them. At the heart of DeepFix is a multi-layered sequence-to-sequence neural network with attention which is trained to predict erroneous program locations along with the required correct statements. On a set of 6971 erroneous C programs written by students for 93 programming tasks, DeepFix could fix 1881 (27%) programs completely and 1338 (19%) programs partially.

3.4 Prex: Finding Guidance for Forward and Backward Porting of Linux Device Drivers


Julia Lawall (INRIA – Paris, FR)

License  Creative Commons BY 3.0 Unported license
© Julia Lawall

A device driver forms the glue between a device and an operating system kernel. When the operating system kernel changes, the device driver has to change as well. Our goal is to automate this kind of forward porting, and analogously backwards porting, focusing on drivers for the Linux kernel. We propose a three step approach, based on 1) compilation of the driver with the target kernel to identify incompatibilities, 2) collection of examples of how to fix these incompatibilities from the commits stored in the revision control system, and 3) generalization of the identified examples to produce change rules appropriate for porting the driver to the target kernel. In this talk, we present the tools gcc-reduce that have been designed to carry out the first two steps. The third step remains future work. Our approach effectively exploits the specific nature of the driver porting problem: the device is fixed and so the required changes are in the interface with the kernel, limiting the kinds of changes required, and many drivers supporting devices with a similar functionality interact with the kernel in a similar way, implying that porting examples are available.

3.5 Automated Inference of Code Transforms and Search Spaces for Patch Generation


Fan Long (MIT – Cambridge, US)

License  Creative Commons BY 3.0 Unported license
© Fan Long

We present a new system, Genesis, that processes sets of human patches to automatically infer code transforms and search spaces for automatic patch generation. We present results that characterize the effectiveness of the Genesis inference algorithms and the resulting complete Genesis patch generation system working with real-world patches and errors collected from top 1000 github Java software development projects. To the best of our knowledge, Genesis is the first system to automatically infer patch generation transforms or candidate patch search spaces from successful patches.

3.6 Learning-based Program Repair

Fan Long (MIT – Cambridge, US)


License  Creative Commons BY 3.0 Unported license
© Fan Long

Code learning and transfer techniques have been recently adopted by many successful automatic patch generation systems to improve patch generation results. This talk presents an overview of these two kinds of techniques. A code learning technique learns useful human knowledge from a training set of past human patches. The technique then applies the learned knowledge either to prioritize correct patches ahead in a patch generation search space or to

infer productive mutation transforms to form the search space. A code transfer technique extracts useful program logic from existing code of a donor program or examples of Q&A websites. It then converts the extracted logic to a patch for a recipient program.

3.7 ASTOR: A Program Repair Library for Java

Matías Sebastián Martínez (University of Valenciennes, FR)

License  Creative Commons BY 3.0 Unported license
© Matías Sebastián Martínez

During the last years, the software engineering research community has proposed approaches for automatically repairing software bugs. Unfortunately, many software artifacts born from this research are not available for repairing Java programs. To reimplement those approaches from scratch is costly. To facilitate experimental replications and comparative evaluations, we present Astor, a publicly available program repair library that includes the implementation of three notable repair approaches (jGenProg2, jKali and jMutRepair). We envision that the research community will use Astor for setting up comparative evaluations and explore the design space of automatic repair for Java. Astor offers researchers ways to implement new repair approaches or to modify existing ones. Astor repairs in total 33 real bugs from four large open source projects.

3.8 Combining syntactic and semantic repair


Sergey Mechtaev (National University of Singapore, SG)

License  Creative Commons BY 3.0 Unported license
© Sergey Mechtaev

Test-driven automated program repair approaches traverse huge search spaces to generate fixes. Several search space prioritization heuristics have been proposed to increase the probability of finding correct repairs. Although existing systems could generate correct fixes for large real-world projects, they suffer from limitations of the search space exploration algorithms. First, current techniques may omit high quality patches during exploration, which results in generation of suboptimal low quality repairs. Second, current techniques are able to explore only relatively small search spaces and, therefore, fix only a small number of defects. We propose a synergy of syntax-based and semantics-based patch generation methods that explicitly generates a search space and semantically partitions it during test execution. The proposed algorithm is able to efficiently traverse the search spaces in an arbitrary order. As a result, our technique is the first that guarantees to the most reliable patch (global maximum) in the search space according to a given static prioritization strategy and yet scales to large search spaces. Evaluation on large real-world subjects revealed that the proposed algorithm generates more repairs, more repairs equivalent to human patches, and find repairs faster compared to previous techniques. Apart from that, the algorithm and the design of the search space enable our system to traverse the search space faster than existing systems with explicit search space representation.

3.9 Antifragile Software and Correctness Attraction

Martin Monperrus (University of Lille & INRIA, FR)

License  Creative Commons BY 3.0 Unported license
© Martin Monperrus

Can the execution of a software be perturbed without breaking the correctness of the output? We present a novel protocol to answer this rarely investigated question. In an experimental study, we observe that many perturbations do not break the correctness in ten subject programs. We call this phenomenon “correctness attraction”. The uniqueness of this protocol is that it considers a systematic exploration of the perturbation space as well as perfect oracles to determine the correctness of the output. To this extent, our findings on the stability of software under execution perturbations have a level of validity that has never been reported before in the scarce related work. A qualitative manual analysis enables us to set up the first taxonomy ever of the reasons behind correctness attraction.

3.10 Detecting and repairing performance bugs that have non-intrusive fixes


Adrian Nistor (Florida State University – Tallahassee, US)

License  Creative Commons BY 3.0 Unported license
© Adrian Nistor

Performance bugs are programming errors that slow down program execution. Unfortunately, many performance bugs cannot be automatically detected or repaired by existing techniques. In this talk I will present Caramel, a novel static analysis technique that detects and then automatically repairs performance bugs that have non-intrusive fixes likely to be adopted by developers. 116 of the bugs found and repaired by Caramel in 15 popular applications (e.g., Chrome, Mozilla, Tomcat, Lucene, Groovy, GCC, MySQL, etc) have already been fixed by developers based on our bug reports.

3.11 Automated Software Transplantation

Justyna Petke (University College London, GB)

License  Creative Commons BY 3.0 Unported license
© Justyna Petke

Genetic Improvement is the application of evolutionary and search-based optimisation methods to the improvement of existing software. For example, it may be used to automate the process of bug-fixing or execution time optimisation. In this talk I present another application of genetic improvement, namely automated software transplantation. While we do not claim automated transplantation is now a solved problem, our results are encouraging: we report that in 12 of 15 experiments, involving 5 donors and 3 hosts (all popular real-world systems), we successfully autotransplanted new functionality from the donor program to the host and passed all regression tests. Autotransplantation is also already useful: in 26 hours computation time we successfully autotransplanted the H.264 video encoding functionality from the x264 system to the VLC media player; compare this to upgrading x264 within VLC, a task that we estimate, from VLC’s version history, took human programmers an average of 20 days of elapsed, as opposed to dedicated, time.

3.12 Understanding and Automatically Preventing Injection Attacks on Node.js

Michael Pradel (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license
© Michael Pradel

The Node.js ecosystem has led to the creation of many modern applications, such as server-side web applications and desktop applications. Unlike client-side JavaScript code, Node.js applications can interact freely with the operating system without the benefits of a security sandbox. The complex interplay between Node.js modules leads to subtle injection vulnerabilities being introduced across module boundaries. This talk presents a large-scale study across 235,850 Node.js modules to explore such vulnerabilities. We show that injection vulnerabilities are prevalent in practice, both due to `eval`, which was previously studied for browser code, and due to the powerful `exec` API introduced in Node.js. Our study shows that thousands of modules may be vulnerable to command injection attacks and that even for popular projects it takes long time to fix the problem. Motivated by these findings, we present `Synode`, an automatic mitigation technique that combines static analysis and runtime enforcement of security policies for allowing vulnerable modules to be used in a safe way. The key idea is to statically compute a template of values passed to APIs that are prone to injections, and to synthesize a grammar-based runtime policy from these templates. Our mechanism does not require the modification of the Node.js platform, is fast (sub-millisecond runtime overhead), and protects against attacks of vulnerable modules while inducing very few false positives (less than 10%).

3.13 Anti-patterns in Search-Based Program Repair


Mukul Prasad (Fujitsu Labs of America Inc. – Sunnyvale, US)

License  Creative Commons BY 3.0 Unported license
© Mukul Prasad

Search-based program repair automatically searches for a program fix within a given repair space. This may be accomplished by retrofitting a generic search algorithm for program repair as evidenced by the `GenProg` tool, or by building a customized search algorithm for program repair as in `SPR`. Unfortunately, automated program repair approaches may produce patches that may be rejected by programmers, because of which past works have suggested using human-written patches to produce templates to guide program repair. In this work, we take the position that we will not provide templates to guide the repair search because that may unduly restrict the repair space and attempt to overfit the repairs into one of the provided templates. Instead, we suggest the use of a set of anti-patterns — a set of generic forbidden transformations that can be enforced on top of any search-based repair tool. We show that by enforcing our anti-patterns, we obtain repairs that localize the correct lines or functions, involve less deletion of program functionality, and are mostly obtained more efficiently. Since our set of anti-patterns are generic, we have integrated them into existing search based repair tools, including `GenProg` and `SPR`, thereby allowing us to obtain higher quality program patches with minimal effort.

3.14 Semantic Techniques for Program Repair

Abhik Roychoudhury (National University of Singapore, SG)

License  Creative Commons BY 3.0 Unported license
© Abhik Roychoudhury

In this talk, I will first recapitulate briefly the challenges in automated program repair. I will then move to discuss semantic analysis techniques for program repair, and position them with respect generate and validate approaches to program repair. I will conclude the talk by discussing how semantic approaches can be combined with generate and validate approaches.

3.15 Automated techniques for fixing performance issues in JavaScript applications

Marija Selakovic (TU Darmstadt, DE)

License  Creative Commons BY 3.0 Unported license
© Marija Selakovic

Many programs suffer from performance problems, but unfortunately, finding and fixing such problems is a cumbersome and time-consuming process. My work focuses on JavaScript, for which little is known about performance issues and how developers address them. To address these questions, I present the empirical study of 98 reproduced performance-related issues from 16 popular JavaScript projects. The findings illustrate that developers optimize their code with relatively simple code changes and that the most common root cause of JavaScript performance issues is the inefficient usage of native and third-party APIs. To help developers find and fix performance problems related to API usages, I present an approach for finding conditionally equivalent APIs and detecting the usages of an API that can be replaced by an equivalent and more efficient alternative for a given input. The approach is based on two-phases dynamic analysis of API usages in web applications. In the first phase, the analysis detects potentially equivalent methods based on their type signatures and name equivalence. In the second phase, the analysis executes these methods with all observed inputs, approximates their execution times and derives an equivalence condition. Finally, the analysis points to all code locations that can be optimized by using more efficient API and suggests a refactoring to the developers.

3.16 I Get by With a Little Help From My Friends: Crowdsourcing Program Repair

Kathryn T. Stolee (North Carolina State University – Raleigh, US)


License  Creative Commons BY 3.0 Unported license
© Kathryn T. Stolee

Regular expressions are commonly used in source code, yet developers find them hard to read, hard to write, and hard to compose. Motivated by the prevalence of regular expression usage in practice and the number of bug reports related to regular expressions, I propose several future directions for studying regular expressions, including error classification, test coverage, test input generation, reuse, and automated program repair. The repair strategies

can work in the presence or absence of fault localization, and with or without test cases. I conclude by discussing the potential impact of integrating regex support into automated program repair approaches.

3.17 Towards Trustworthy Program Repair

Yingfei Xiong (Peking University, CN)

License  Creative Commons BY 3.0 Unported license
© Yingfei Xiong

Many different approaches have been proposed for automatic program repair, but the precision and recall of current techniques are not satisfactory. This talk will introduce our work exploring the possibilities of improving precision and recall. First, we did a study of manual program repair, and the results show that human developer indeed can achieve high precision and recall under the same setting of automatic program repair. Second, we proposed two techniques, mining QA site and statistical condition synthesis, mainly for improving precision. Both techniques achieve a precision of around 80%.

3.18 How Developers Diagnose and Repair Software Bugs (and what we can do about it)

Andreas Zeller (Universität des Saarlandes, DE)

License  Creative Commons BY 3.0 Unported license
© Andreas Zeller

How do practitioners debug computer programs? In a retrospective study with 180 respondents and an observational study with 12 practitioners, we collect and discuss data on how developers spend their time on diagnosis and fixing bugs, with key findings on tools and strategies used, as well as highlighting the need for automated assistance. To facilitate and guide future research, we provide DBGBENCH, a highly usable debugging benchmark providing fault locations, patches and explanations for common bugs as provided by the practitioners.

3.19 Automated Test Reuse via Code Transplantation

Tianyi Zhang (UCLA, US)

License  Creative Commons BY 3.0 Unported license
© Tianyi Zhang

Code clones are common in software. When applying similar edits to clones, developers often find it difficult to examine the runtime behavior of clones. The problem is exacerbated when some clones are tested, while their counterparts are not. To reuse tests for similar but not identical clones, Grafter transplants one clone to its counterpart by (1) identifying variations in identifier names, types, and method call targets, (2) resolving compilation errors caused by such variations through code transformation, and (3) inserting stub code to transfer input data and intermediate output values for examination. To help developers cross-check

behavioral consistency between clones, Grafter supports fine-grained differential testing at both the test outcome level and the intermediate program state level.

In our evaluation on three open source projects, Grafter successfully reuses tests in 94% of clone pairs without inducing build errors, demonstrating its automated test transplantation capability. To examine the robustness of Grafter, we automatically insert faults using a mutation testing tool, Major, and check for behavioral consistency using Grafter. Compared with a static cloning bug finder, Grafter detects 31% more mutants using the test-level comparison and almost 2X more using the state-level comparison. This result indicates that GRAFTER should effectively complement static cloning bug finders.

Participants

- Yuriy Brun
University of Massachusetts – Amherst, US
- Celso G. Camilo-Junior
Federal University of Goiás, BR
- Jonathan Dorn
University of Virginia – Charlottesville, US
- Lars Grunske
HU Berlin, DE
- Ciera Jaspán
Google Inc. – Mountain View, US
- Aditya Kanade
Indian Institute of Science – Bangalore, IN
- Sarfraz Khurshid
University of Texas – Austin, US
- Dongsun Kim
University of Luxembourg, LU
- Sunghun Kim
HKUST – Kowloon, HK
- Julia Lawall
INRIA – Paris, FR
- Claire Le Goues
Carnegie Mellon University – Pittsburgh, US
- Fan Long
MIT – Cambridge, US
- Matías Sebastián Martínez
University of Valenciennes, FR
- Sergey Mechtaev
National University of Singapore, SG
- Martin Monperrus
University of Lille & INRIA, FR
- Adrian Nistor
Florida State University – Tallahassee, US
- Alessandro Orso
Georgia Institute of Technology – Atlanta, US
- Justyna Petke
University College London, GB
- Michael Pradel
TU Darmstadt, DE
- Mukul Prasad
Fujitsu Labs of America Inc. – Sunnyvale, US
- Abhik Roychoudhury
National University of Singapore, SG
- Marija Selakovic
TU Darmstadt, DE
- Kathryn T. Stolee
North Carolina State University – Raleigh, US
- David R. White
University College London, GB
- Yingfei Xiong
Peking University, CN
- Andreas Zeller
Universität des Saarlandes, DE
- Tianyi Zhang
UCLA, US

