

# Automated Programming and Program Repair

Claire Le Goues<sup>\*1</sup>, Michael Pradel<sup>\*2</sup>, Abhik Roychoudhury<sup>\*3</sup>, and Shin Hwei Tan<sup>\*4</sup>

1 Carnegie Mellon University – Pittsburgh, US. [clegoues@cs.cmu.edu](mailto:clegoues@cs.cmu.edu)

2 Universität Stuttgart, DE. [michael@binaervarianz.de](mailto:michael@binaervarianz.de)

3 National University of Singapore, SG. [abhik.roychoudhury@gmail.com](mailto:abhik.roychoudhury@gmail.com)

4 Concordia University – Montreal, CA. [shinhwei.tan@concordia.ca](mailto:shinhwei.tan@concordia.ca)

---

## Abstract

The Dagstuhl Seminar 24431 on “Automated Programming and Program Repair” brought together 33 researchers from academia and industry to explore the intersection of automated code generation and program repair. Over five days (October 21–25, 2024), participants discussed advances in large language models (LLMs) for code generation, the role of automated program repair in improving generated code, and challenges in deploying these technologies in real-world software development. The seminar featured over 20 talks and three panel discussions on topics such as benchmarks for LLM-generated code, trust in automated programming, and the broader applications of LLMs beyond coding assistance. Key outcomes included identifying critical challenges in benchmarking, evaluation criteria, and developer adoption of automated repair techniques, fostering future collaborations and actionable research directions in the field.

**Seminar** October 20–25, 2024 – <https://www.dagstuhl.de/24431>

**2012 ACM Subject Classification** Software and its engineering → Automatic programming;

Software and its engineering → Software testing and debugging

**Keywords and phrases** Auto-coding, Large Language Models, Automated Program Repair, Program Synthesis, Trustworthy Software

**Digital Object Identifier** 10.4230/DagRep.14.10.39

## 1 Executive Summary

*Shin Hwei Tan (Concordia University – Montreal, CA)*

**License**  Creative Commons BY 4.0 International license  
© Shin Hwei Tan

Automated tools that generate and improve code promise to fundamentally change software development. For example, there is a recent trend towards automated code generation from large language models, as evidenced by the capabilities of Codex/Copilot, ChatGPT, and GPT-4. These models, and other techniques, such as search-based and semantic analysis-based techniques, have the potential to automate significant parts of today’s software development process. In particular, there are promising techniques for automated programming and automated program repair. Automated programming refers to techniques that suggest newly written code, e.g., in the form of code completion tools. The capabilities of such tools have increased from moderately successful single-token predictions just a few years ago to predicting entire functions with relatively high accuracy. Techniques for automated programming include large language models that predict code based on natural language specifications of the intended behavior.

---

\* Editor / Organizer



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Automated Programming and Program Repair, *Dagstuhl Reports*, Vol. 14, Issue 10, pp. 39–57

Editors: Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Shin Hwei Tan



Dagstuhl Reports

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

Automated program repair refers to a suite of techniques for automated rectification of errors or vulnerabilities in programs. Automated program repair technologies were originally developed for reducing the debugging effort for manually written code. In other words, automated program repair technologies are meant to boost developer productivity in locating and fixing errors in manually written code. Automated programming and automated program repair strongly overlap in terms of their goals and the techniques used to achieve these goals. Both streams of research aim at generating correct source code while having to cope with limited knowledge of the behavior this source code is meant to have. Since formal specifications of correct program behavior are typically not available, both techniques try to infer specifications from various program artifacts, such as large code corpora, past program versions, natural language documentation, or various executions of the program. To address the challenge of predicting likely correct code, both streams of research combine techniques from machine learning, search-based approaches, and semantic code analysis.

Despite these similarities, the subcommunities working on automated programming and automated program repair are only partially aware of each other's most recent techniques. This seminar set out to explore the intersection of these two fields in order to foster collaborations between them. For example, to discuss recent work and potential future work in the following directions: (1) Apply program repair to fix code generated by code completion models. The code generated by large language models often leaves significant room for improvement in terms of correctness, thereby raising the question whether automated program repair techniques can be used for last-mile improvement of code that was automatically generated by large language models. (2) Apply the generate-and-validate paradigm from program repair to the code completion problem. For example, such techniques can repeatedly generate code completion candidates and validate them by running test suites. (3) Apply language model-based code generators to the program repair problem. Once the location of a bug has been (heuristically) determined, large language models can predict candidate code snippets for replacing the incorrect code. (4) Use the ability of large language models to infer the intended behavior of code from natural language information embedding in the code. For example, we plan to discuss techniques that specify the intended behavior in the form of assertions or test cases, which can then guide automated program repair. (5) In addition to predicting (fixed) code, generate evidence that the final code is trustworthy. Such evidence may take the form of tests generated along with the code, or other certificates obtained from formal reasoning.

Thus, to discuss topics at the intersection of automated programming and program repair, we had Dagstuhl Seminar 24431. In a five-day seminar with 33 participants from both academia and industries (e.g., Microsoft and Google), we held a series of talks and three panel discussions. The seminar concluded with more than 20 talks and three panel discussions. Overall, the seminar stimulated quite a few discussions where researchers initiated some future research directions and potential international collaborations.

Before the seminar, all participants had received an invitation to give a talk of a flexible duration (i.e., lightning update that is around 5 minutes, short talk that is around 10 minutes or long talk that is around 25 minutes). More than 20 participants replied positively to the invitation, resulting in a great variety of talks given by many participants. The first day of the seminar (i.e., October 21, 2025) started with an introduction by the organizers and a self-introduction by all the participants. Then, a few short talks and longer talks (more than 25 minutes) were given by participants. The first day ended with a panel discussion on "Benchmarks for LLM Code Generation". On October 22, 2025, there were several talks followed by a panel discussion on "LLM-beyond just coding-assistance". On October 23,

2025, several talks took place in the morning, followed by an excursion to Mettlach and Villa Borg after lunch. On October 24, 2025, a few inspiring talks took place, followed by a panel discussion on “Obstacles for deploying program repair techniques”.

Overall, the seminar has received very positive feedback from the participants both personally and formally (via email). Notably, one participant sent an email to one of the organizers saying that “It was my best Dagstuhl Seminar last October, and I really appreciate your organizing of the seminar once again”, demonstrating that the seminar has been quite successful in leaving a good impression in comparison to other Dagstuhl Seminars that the participants have attended. Meanwhile, a few participants have complimented Dagstuhl on the diversity of the social events held (e.g., excursion, the treetop walk, and sauna), and the babysitting services provided for participants attending the seminar with young children.

In terms of collaborations, there are a few actionable topics for collaborations that have been discussed. An opinion piece of AI Software Engineer titled “AI Software Engineer: Programming with Trust” is now available.<sup>1</sup> Another potential collaboration is a critical review on benchmarks crafted by AI communities (i.e., SWE-Bench). Meanwhile, AutoCodeRover (presented by one of the organizers in Dagstuhl), which was an NUS spinoff, has on February 19, 2025, been officially acquired by SonarSource, a leader in code quality via its static analysis solutions.

The seminar focused on the following key themes:

- Topics at the intersection of automated programming and automated program repair, analyzing progress in both fields.
- Understanding common mistakes in automatically generated code.
- Discussing the theme of “**Trusted Automated Programming**”, which focuses on:
  - How automatically generated code can be made more trustworthy.
  - How to generate evidence that improvements to auto-generated code maintain trustworthiness.
  - How to decide, based on such evidence, when to incorporate automatically generated code into an existing software project with a stable code-base.
- Important challenges in automated program repair and automated programming in general.
- Using large language models (LLMs) beyond just coding assistance.
- Obstacles in deploying program repair techniques in real-world settings.

The seminar also identified several critical challenges:

1. The problem of curating widely accepted benchmarks for code generation that serve both the Software Engineering and AI communities.
2. The problem of designing evaluation criteria that effectively assess the quality and reliability of auto-generated code.
3. The challenges and opportunities in applying LLM-based techniques beyond traditional automated program repair (APR).
4. The obstacles in training developers to effectively use program repair techniques in real-world software development.

---

<sup>1</sup> <https://arxiv.org/abs/2502.13767>

## 2 Table of Contents

### Executive Summary

<i>Shin Hwei Tan</i> . . . . .	39
--------------------------------	----

### Overview of Talks

Automatic Semantic Augmentation of Language Model Prompts <i>Earl T. Barr</i> . . . . .	44
RepairAgent: An Autonomous, LLM-Based Agent for Program Repair <i>Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel</i> . . . . .	44
Ya'll are APRing the wrong thing <i>Yuriy Brun</i> . . . . .	45
Reflections on Automated Programming / Input Repair: Can LLMs help? <i>Cristian Cadar</i> . . . . .	45
Fault Localization (LLM + Heuristics): initial results <i>Celso G. Camilo-Junior</i> . . . . .	46
Reinventing ourselves <i>Satish Chandra</i> . . . . .	46
Automated Software A11Y Repair <i>Chunyang Chen</i> . . . . .	47
Enhancing Fault Localization with LLM Agents and Self-Reflection <i>Tse-Hsun Chen</i> . . . . .	47
Code from LLMs: Use, modify, or discard? <i>Premkumar T. Devanbu</i> . . . . .	48
Automated Scientific Debugging with LLMs <i>Sungmin Kang</i> . . . . .	48
Proactive Debugging <i>Dongsun Kim</i> . . . . .	48
Energy Consumption of Automated Program Repair <i>Matías Martínez</i> . . . . .	49
Fact Selection Problem in LLM Based Program Repair <i>Nikhil Parasaram</i> . . . . .	49
ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution <i>Michael Pradel</i> . . . . .	50
Testing, Testing, 1-2-3: Test generation with LLMs <i>Nikitha Rao</i> . . . . .	51
Assured Automatic Programming via Large Language Models <i>Abhik Roychoudhury, Andreea Costea</i> . . . . .	51
AutoCodeRover: Autonomous Program Improvement <i>Abhik Roychoudhury</i> . . . . .	52
RepairBench: Leaderboard of Frontier Models for Program Repair <i>André Silva</i> . . . . .	52

Debugging and Fixing Fairness Issues in Machine Learning <i>Gang (Gary) Tan</i> . . . . .	53
More Data or More Domain Knowledge? – For LLM-based Automatic Programming and Repair <i>Lin Tan</i> . . . . .	53
Neural Code Generation Models with Programming Language Knowledge <i>Yingfei Xiong</i> . . . . .	54
Automated Program Repair from Fuzzing Perspective <i>Jooyong Yi</i> . . . . .	54
Automated Programming in the Era of Large Language Models <i>Lingming Zhang</i> . . . . .	54
<b>Panel discussions</b>	
Benchmarks for LLM Code Generation <i>Satish Chandra, Premkumar T. Devanbu, Martin Monperrus, Gustavo Soares, and Lin Tan</i> . . . . .	55
LLM-beyond just coding-assistance <i>Ahmed E. Hassan, Lin Tan, and Shin Hwei Tan</i> . . . . .	55
Obstacles for deploying program repair techniques <i>Fernanda Madeiral, Premkumar T. Devanbu, and Gustavo Soares</i> . . . . .	56
<b>Participants</b> . . . . .	57

### 3 Overview of Talks

#### 3.1 Automatic Semantic Augmentation of Language Model Prompts

Earl T. Barr (*University College London, GB*)

**License** © Creative Commons BY 4.0 International license  
© Earl T. Barr

**Joint work of** Toufique Ahmed, Kunal Suresh Pai, Premkumar T. Devanbu, Earl T. Barr  
**Main reference** Toufique Ahmed, Kunal Suresh Pai, Premkumar T. Devanbu, Earl T. Barr: “Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization)”, in Proc. of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, pp. 220:1–220:13, ACM, 2024.  
**URL** <https://doi.org/10.1145/3597503.3639183>

Researchers are still learning how to best “program” LLMs via prompt engineering. We start with the intuition that developers tend to consciously and unconsciously collect semantics facts, from code, while working. Most are shallow, simple facts arising from a quick read. One might assume that LLMs are implicitly capable of performing simple “code analysis” and extracting such information: but are they, really? If not, could explicitly adding this information help? Our goal here is to investigate this question and evaluate whether automatically augmenting an LLM’s prompt with semantic facts explicitly, actually helps. We find that adding semantic facts to the prompt actually does help! This approach improves performance on the code summarization and completion tasks in several different settings suggested by prior work, including for three different Large Language Models. In most cases, we see improvements, as measured by a range of commonly-used metrics.

#### 3.2 RepairAgent: An Autonomous, LLM-Based Agent for Program Repair

Islem Bouzenia (*Universität Stuttgart, DE*), Premkumar T. Devanbu (*University of California – Davis, US*), Michael Pradel (*Universität Stuttgart, DE*)

**License** © Creative Commons BY 4.0 International license  
© Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel

Automated program repair has emerged as a powerful technique to mitigate the impact of software bugs on system reliability and user experience. This paper introduces RepairAgent, the first work to address the program repair challenge through an autonomous agent based on a large language model (LLM). Unlike existing deep learning-based approaches, which prompt a model with a fixed prompt or in a fixed feedback loop, our work treats the LLM as an agent capable of autonomously planning and executing actions to fix bugs by invoking suitable tools. RepairAgent freely interleaves gathering information about the bug, gathering repair ingredients, and validating fixes, while deciding which tools to invoke based on the gathered information and feedback from previous fix attempts. Key contributions that enable RepairAgent include a set of tools that are useful for program repair, a dynamically updated prompt format that allows the LLM to interact with these tools, and a finite state machine that guides the agent in invoking the tools. Our evaluation on the popular Defects4J dataset demonstrates RepairAgent’s effectiveness in autonomously repairing 164 bugs, including 39 bugs not fixed by prior techniques. Interacting with the LLM imposes an average cost of 270,000 tokens per bug, which, under the current pricing of OpenAI’s GPT-3.5 model, translates to 14 cents per bug. To the best of our knowledge, this work is the first to present an autonomous, LLM-based agent for program repair, paving the way for future agent-based techniques in software engineering.

### 3.3 Ya'll are APRing the wrong thing

*Yuriy Brun (University of Massachusetts Amherst, US)*

**License** © Creative Commons BY 4.0 International license  
 © Yuriy Brun  
**Joint work of** Yuriy Brun, Emily First, Arjun Guha, Zhanna Kaufman, Alex Sanchez-Stern, Abhishek Varghese, Saketh Kasibatla, Arpan Agrawal, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Dylan Zhang, Sorin Lerner, Talia Ringer  
**Main reference** Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Dylan Zhang, Talia Ringer, Yuriy Brun: “QEDCartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning”, in Proc. of the 47th International Conference on Software Engineering (ICSE), pp. 405–418, 2025.  
**URL** <https://doi.org/10.1109/ICSE55347.2025.00033>

Automated program repair (APR) has advanced program synthesis technology but is limited by the problem of weak specifications. The domain of proof synthesis for formal verification has many of the same challenges as APR but has access to a strong oracle – the theorem prover – to determine proof correctness. This oracle helps overcome the overfitting problem for proof synthesis. We have developed several tools for synthesizing proofs from scratch using natural language processing, including TacTok, Passport, Diva, Baldur, and Cobblestone.

#### References

- 1 First, Emily and Brun, Yuriy and Guha, Arjun, TacTok: Semantics-aware proof synthesis, OOPSLA 2020
- 2 Sanchez-Stern, Alex and First, Emily and Zhou, Timothy and Kaufman, Zhanna and Brun, Yuriy and Ringer, Talia, Passport: Improving automated formal verification using identifiers, TOPLAS23
- 3 Sanchez-Stern, Alex and Varghese, Abhishek and Kaufman, Zhanna and Zhang, Dylan and Ringer, Talia and Brun, Yuriy, QEDCartographer: Automating formal verification using reward-free reinforcement learning, ICSE 25, To Appear
- 4 First, Emily and Brun, Yuriy, Diversity-driven automated formal verification, ICSE 2022
- 5 First, Emily and Rabe, Markus N and Ringer, Talia and Brun, Yuriy, Baldur: Whole-proof generation and repair with large language models, FSE 2023
- 6 First, Emily and Brun, Yuriy, Diversity-driven automated formal verification, ICSE 2022
- 7 Brun, Yuriy, Demo for Proofster: Automated Formal Verification, <https://www.youtube.com/watch?v=xQAi66lRfwI>

### 3.4 Reflections on Automated Programming / Input Repair: Can LLMs help?


*Cristian Cadar (Imperial College London, GB)*

**License** © Creative Commons BY 4.0 International license  
 © Cristian Cadar  
**Joint work of** Tomasz Kuchta, Cristian Cadar, Miguel Castro, Manuel Costa  
**Main reference** Tomasz Kuchta, Cristian Cadar, Miguel Castro, Manuel Costa: “Docovery: toward generic automatic document recovery”, in Proc. of the ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden – September 15 – 19, 2014, pp. 563–574, ACM, 2014.  
**URL** <https://doi.org/10.1145/2642937.2643004>

In my talk, I first reflect on the challenges of integrating automated programming into software projects maintained by humans. I then introduce the problem of input repair, particularly in the context of complex documents and document processors, with the goal of discussing the applicability of modern LLM-based code repair techniques.

### 3.5 Fault Localization (LLM + Heuristics): initial results

*Celso G. Camilo-Junior (Federal University of Goiás, BR)*

License  Creative Commons BY 4.0 International license  
© Celso G. Camilo-Junior

The task of Fault Localization is an important part of the software debugging process, as it requires a lot of effort and investment. Additionally, it directly impacts the quality of the produced code.


With the emergence of LLMs and their potential, there is an opportunity to apply them to Fault Localization as well. However, discovering the best way to integrate known techniques with LLMs is a challenge.

Thus, this presentation shows some initial results on how the use of LLMs for Fault Localization, utilizing a few facts, can be reasonable, and which scenarios still face challenges for applying more complex methods. Moreover, an architecture was presented that aims to enrich the necessary inputs, such as better problem descriptions, for the optimal performance of these tools.

The initial results show that, for some simpler problems, using a generic LLM (GPT-3.5 and Llama 3) with a good human description or an artificial description yields reasonable results. However, in more complex scenarios, the combination of LLM with heuristics (spectrum-based) shows more promising outcomes.

### 3.6 Reinventing ourselves

*Satish Chandra (Google – Mountain View, US)*

License  Creative Commons BY 4.0 International license  
© Satish Chandra

In this talk, I will cover a few related themes. First, I will share my views on how modern LLMs are upending the field of software engineering with capabilities that only a few years ago required creative solutions. Next, I'll describe how at Google we have been working on weaving AI capabilities in developer workflows, how we collect developer data, and how we prioritize our work (based on the a recent blog post <https://research.google/blog/ai-in-software-engineering-at-google-progress-and-the-path-ahead/>). I will then change tracks a bit and discuss the importance of evals for software engineering tasks; the more real the better. Finally, I will segue into our preliminary investigation into understanding our own bug database at Google, and how it may compare in distribution to SWEBench. I'll share our preliminary experience with agentic ways of resolving these internal bugs.

### 3.7 Automated Software A11Y Repair

*Chunyang Chen (TU München – Heilbronn, DE)*

**License** © Creative Commons BY 4.0 International license  
© Chunyang Chen

**Main reference** Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, Qing Wang: “Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions”, in Proc. of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, pp. 100:1–100:13, ACM, 2024.  
**URL** <https://doi.org/10.1145/3597503.3639180>

As one of the major infrastructures of society, software must be accessible to everyone, regardless of their physical abilities or socioeconomic status. However, the stark reality is that much software remains inaccessible or challenging to use for individuals with disabilities, creating a significant digital divide. To repair those accessibility issues, I will present our works using machine learning to automatically repair those issues i.e., generating missing labels for interactive icons in the GUI based on computer vision, and generating meaningful hint-text for text blanks in mobile applications with LLM.

### 3.8 Enhancing Fault Localization with LLM Agents and Self-Reflection

*Tse-Hsun Chen (Concordia University – Montreal, CA)*

**License** © Creative Commons BY 4.0 International license  
© Tse-Hsun Chen

**Joint work of** Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, Shaowei Wang  
**Main reference** Md Nakhla Rafi, Dong Jae Kim, Tse-Hsun Chen, Shaowei Wang: “Enhancing Fault Localization Through Ordered Code Analysis with LLM Agents and Self-Reflection”, CoRR, Vol. abs/2409.13642, 2024.  
**URL** <https://arxiv.org/abs/2409.13642>

Locating and fixing software faults is time-consuming and resource-intensive. Traditional methods like Spectrum-Based Fault Localization (SBFL) suffer from low accuracy, while learning-based approaches require large datasets and are computationally expensive. Recent advancements in Large Language Models (LLMs) show promise in improving fault localization, but they still face challenges such as token limits and difficulties with large projects. To address this, we introduce LLM4FL, an LLM-agent-based approach that combines SBFL with a divide-and-conquer strategy. By using multiple LLM agents and prompt chaining, LLM4FL navigates codebases and localizes faults more effectively. In tests using Defects4J, LLM4FL outperformed AutoFL by 19.27% in Top-1 accuracy and surpassed state-of-the-art methods like DeepFL and Grace, without task-specific training. Coverage splitting and method ordering further boosted accuracy by up to 22%.

### 3.9 Code from LLMs: Use, modify, or discard?

*Premkumar T. Devanbu (University of California – Davis, US)*

**License** © Creative Commons BY 4.0 International license  
© Premkumar T. Devanbu

**Joint work of** Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Premkumar T. Devanbu, Toufique Ahmed

Code generated by language models is often flawed. And yet, LLMs can produce a lot of useful code. How is a developer to decide whether a given generated fragment of code should be used or not? We consider this to be a joint human-AI decision problem, where it vitally important for the AI (LLM) to reveal to the AI some trustworthy indication of the expected likelihood of the code being correct.

#### References

- 1 Spiess, Claudio and Gros, David and Pai, Kunal Suresh and Pradel, Michael and Rabin, Md Rafiqul Islam and Alipour, Amin and Jha, Susmit and Devanbu, Prem and Ahmed, Toufique, Calibration and correctness of language models for code, ICSE 2025, *to appear*

### 3.10 Automated Scientific Debugging with LLMs

*Sungmin Kang (KAIST – Daejeon, KR)*

**License** © Creative Commons BY 4.0 International license  
© Sungmin Kang

**Joint work of** Sungmin Kang, Bei Chen, Shin Yoo, Jian-Guang Lou

**Main reference** Sungmin Kang, Bei Chen, Shin Yoo, Jian-Guang Lou: “Explainable Automated Debugging via Large Language Model-driven Scientific Debugging”, CoRR, Vol. abs/2304.02195, 2023.

**URL** <https://doi.org/10.48550/ARXIV.2304.02195>

Existing work on the application of automated program repair (APR) techniques in industry suggest that providing explanations for automatically generated patches could help developer adoption of such tools. We first define criteria for satisfying explanations of patches, then propose the Automated Scientific Debugging (AutoSD) technique, inspired by human developer debugging frameworks, to uncover facts about the debugging scenario and integrate these facts with hypotheses about what is causing the bug. We present our empirical results demonstrating that AutoSD achieves comparable repair performance with other techniques, and that explanations could help developers assess the correctness of patches generated by AutoSD.

### 3.11 Proactive Debugging

*Dongsun Kim (Kyungpook National University, KR)*

**License** © Creative Commons BY 4.0 International license  
© Dongsun Kim

Automated program repair (APR) techniques successfully fixed programs bugs, but it still follows the process of classical debugging, which is not quite different from a manual debugging process. Although the techniques are fully or partially automated, we have to wait for a bug report or symptoms, reproduce it, localize it, and fix it. However, it might not be necessary to follow the classical and reactive process. What if we can proactively change the source code first to fix a bug?

This talk addresses a series of our initial work on proactive debugging applied to specific bug types, such as memory leaks in single-page web applications and blocking-call bugs in reactive applications. In particular, the talk demonstrates the effectiveness of pattern-based patch generation as the first step of proactive debugging. To show the effectiveness of proactive debugging, the work conducts a series of experiments. The results show that the proactive process significantly helps the developers address bugs in real software systems. The talk illustrates further challenges to improve the activities for proactive debugging, such as fully automated evidence collection and patch generation techniques.

### 3.12 Energy Consumption of Automated Program Repair

*Matías Martínez (UPC Barcelona Tech, ES)*

**License** © Creative Commons BY 4.0 International license  
© Matías Martínez

**Joint work of** Matías Martínez, Silverio Martínez-Fernández, Xavier Franch

**Main reference** Matias Martinez, Silverio Martínez-Fernández, Xavier Franch: “Energy Consumption of Automated Program Repair”, in Proc. of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2024, Lisbon, Portugal, April 14-20, 2024, pp. 358–359, ACM, 2024.

**URL** <https://doi.org/10.1145/3639478.3643114>

Automated program repair (APR) aims to automatize the process of repairing software bugs in order to reduce the cost of maintaining software programs. Moreover, the success (given by the accuracy metric) of APR approaches has increased in recent years. However, no previous work has considered the energy impact of repairing bugs automatically using APR. The field of green software research aims to measure the energy consumption required to develop, maintain, and use software products. This paper combines, for the first time, the APR and Green software research fields. We have as main goal to define the foundation for measuring the energy consumption of the APR activity. We measure the energy consumption of ten traditional program repair tools for Java and ten fine-tuned Large-Language Models (LLM) on source code trying to repair real bugs from Defects4J, a set of real buggy programs. The initial results from this experiment show the existing trade-off between energy consumption and the ability to correctly repair bugs: Some APR tools are capable of achieving higher accuracy by spending less energy than other tools.

### 3.13 Fact Selection Problem in LLM Based Program Repair

*Nikhil Parasaram (University College London, GB)*

**License** © Creative Commons BY 4.0 International license  
© Nikhil Parasaram

**Joint work of** Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, Sergey Mechtaev

**Main reference** Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, Sergey Mechtaev: “The Fact Selection Problem in LLM-Based Program Repair”, CoRR, Vol. abs/2404.05520, 2024.

**URL** <https://doi.org/10.48550/ARXIV.2404.05520>

Recent research shows that including bug-related facts (categories of the information), such as stack traces and GitHub issues, enhances LLMs’ bug-fixing abilities. To determine the optimal facts and quantity, we studied over 19K prompts with various fact combinations to repair 314 BugsInPy benchmark bugs. Each fact, from syntactic details to unexplored

semantic information like angelic values, proved beneficial in fixing specific bugs. Notably, effectiveness is non-monotonic: too many facts reduce performance. We defined the fact selection problem to find the optimal facts for each task, developing MANIPLE, a model that selects facts tailored to a given bug, outperforming existing methods and repairing 17% more bugs than the best alternative.

### 3.14 ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution

*Michael Pradel (Universität Stuttgart, DE)*

**License** © Creative Commons BY 4.0 International license  
© Michael Pradel

**Joint work of** Lars Gröninger, Beatriz Souza, Michael Pradel

**Main reference** Lars Gröninger, Beatriz Souza, Michael Pradel: “ChangeGuard: Validating Code Changes via Pairwise Learning-Guided Execution”, CoRR, Vol. abs/2410.16092, 2024.

**URL** <https://doi.org/10.48550/ARXIV.2410.16092>

Code changes are an integral part of the software development process. Many code changes are meant to improve the code without changing its functional behavior, e.g., refactorings and performance improvements. Unfortunately, validating whether a code change preserves the behavior is non-trivial, particularly when the code change is performed deep inside a complex project. This talk presents ChangeGuard, an approach that uses learning-guided execution to compare the runtime behavior of a modified function. The approach is enabled by the novel concept of pairwise learning-guided execution and by a set of techniques that improve the robustness and coverage of the state-of-the-art learning-guided execution technique. Our evaluation applies ChangeGuard to a dataset of 224 manually annotated code changes from popular Python open-source projects and to three datasets of code changes obtained by applying automated code transformations. Our results show that the approach identifies semantics-changing code changes with a precision of 77.1% and a recall of 69.5%, and that it detects unexpected behavioral changes introduced by automatic code refactoring tools. In contrast, the existing regression tests of the analyzed projects miss the vast majority of semantics-changing code changes, with a recall of only 7.6%. We envision our approach being useful for detecting unintended behavioral changes early in the development process and for improving the quality of automated code transformations.

### 3.15 Testing, Testing, 1-2-3: Test generation with LLMs

*Nikitha Rao (Carnegie Mellon University – Pittsburgh, US)*

- License** © Creative Commons BY 4.0 International license  
© Nikitha Rao
- Main reference** Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, Vincent J. Hellendoorn: “CAT-LM Training Language Models on Aligned Code And Tests”, in Proc. of the 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023, pp. 409–420, IEEE, 2023.
- URL** <https://doi.org/10.1109/ASE56229.2023.00193>
- Main reference** Nikitha Rao, Elizabeth Gilbert, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, Sarah Fakhoury: “DiffSpec: Differential Testing with LLMs using Natural Language Specifications and Code Artifacts”, CoRR, Vol. abs/2410.04249, 2024.
- URL** <https://doi.org/10.48550/ARXIV.2410.04249>

LLMs can generate code that is highly similar to that written by humans. However, current models are trained to generate each file separately, as is standard practice in natural language processing. They thus fail to generate meaningful tests. My work leverages software artifacts like code and natural language specifications to make LLM-based test generation more reliable and useful. This includes (1) CAT-LM, a LLM trained to explicitly consider the mapping between code and test files to improve the quality of tests generated. This not only ensures code correctness, but also optimizes for other software testing metrics such as pass/compile rate and code coverage. (2) DiffSpec, a prompt chaining based framework that leverages artifacts like specification documents, bug reports, and code implementations for generating differential tests using LLMs. We evaluate DiffSpec on eBPF and Wasm, and generated over 500 differentiating tests that found real bugs.

### 3.16 Assured Automatic Programming via Large Language Models

*Abhik Roychoudhury (National University of Singapore, SG), Andreea Costea (National University of Singapore, SG)*

- License** © Creative Commons BY 4.0 International license  
© Abhik Roychoudhury, Andreea Costea
- Joint work of** Abhik Roychoudhury, Andreea Costea, Abhishek Kr Singh, Martin Mirchev

With the advent of LLMs in automatic programming, the interest in trusted automatic programming via LLMs increases. Unfortunately, it is difficult to give any guarantees about code generated from LLMs, partly also because a detailed specification of the intended behavior is usually not available. In this talk we show how to alleviate this lack of functionality specifications by aligning automatically generated code via LLMs, automatically generated formal specifications (obtained from natural language using LLMs), as well as tests. The conformance between generated programs, generated specifications, and tests – does not provide absolute guarantees but enhances trust. Establishing such conformance also helps us uncover the likely intended program behavior.

### 3.17 AutoCodeRover: Autonomous Program Improvement

*Abhik Roychoudhury (National University of Singapore, SG)*

**License** © Creative Commons BY 4.0 International license  
© Abhik Roychoudhury

**Joint work of** Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, Abhik Roychoudhury

**Main reference** Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, Abhik Roychoudhury: “AutoCodeRover: Autonomous Program Improvement”, in Proc. of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024, pp. 1592–1604, ACM, 2024.

**URL** <https://doi.org/10.1145/3650212.3680384>

Researchers have made significant progress in automating the software development process in the past decades. Recent progress in Large Language Models (LLMs) has significantly impacted the development process, where developers can use LLM-based programming assistants to achieve automated coding. Nevertheless, software engineering involves the process of program improvement apart from coding, specifically to enable software maintenance (e.g. bug fixing) and software evolution (e.g. feature additions). We propose an automated approach for solving GitHub issues to autonomously achieve program improvement. In our approach called AutoCodeRover, LLMs are combined with sophisticated code search capabilities, ultimately leading to a program modification or patch. In contrast to recent LLM agent approaches from AI researchers and practitioners, our outlook is more software engineering oriented. We work on a program representation (abstract syntax tree) as opposed to viewing a software project as a mere collection of files. Our code search exploits the program structure in the form of classes/methods to enhance LLM’s understanding of the issue’s root cause, and effectively retrieve a context via iterative search. The use of spectrum-based fault localization using tests, further sharpens the context, as long as a test-suite is available. Experiments on SWE-bench-lite (300 real-life GitHub issues) show increased efficacy in solving GitHub issues (19% on SWE-bench-lite), which is higher than the efficacy of the recently reported SWE-agent. In addition, AutoCodeRover achieved this efficacy with significantly lower cost (on average, \$0.43 USD), compared to other baselines. We posit that our workflow enables autonomous software engineering, where, in future, auto-generated code from LLMs can be autonomously improved.

### 3.18 RepairBench: Leaderboard of Frontier Models for Program Repair

*André Silva (KTH Royal Institute of Technology – Stockholm, SE)*

**License** © Creative Commons BY 4.0 International license  
© André Silva

**Joint work of** André Silva, Martin Monperrus

**Main reference** André Silva, Martin Monperrus: “RepairBench: Leaderboard of Frontier Models for Program Repair”, CoRR, Vol. abs/2409.18952, 2024.

**URL** <https://doi.org/10.48550/ARXIV.2409.18952>

AI-driven program repair uses AI models to repair buggy software by producing patches. Rapid advancements in AI surely impact state-of-the-art performance of program repair. Yet, grasping this progress requires frequent and standardized evaluations. We propose RepairBench, a novel leaderboard for AI-driven program repair. The key characteristics of RepairBench are: 1) it is execution-based: all patches are compiled and executed against a test suite, 2) it assesses frontier models in a frequent and standardized way. RepairBench leverages two high-quality benchmarks, Defects4J and GitBug-Java, to evaluate frontier models against real-world program repair tasks. We publicly release the evaluation framework of RepairBench. We will update the leaderboard as new frontier models are released.

### 3.19 Debugging and Fixing Fairness Issues in Machine Learning

*Gang (Gary) Tan (Pennsylvania State University – University Park, US)*

- License** © Creative Commons BY 4.0 International license  
© Gang (Gary) Tan
- Joint work of** Saeid Tizpaz-Niari, Ashish Kumar, Ashutosh Trivedi, Vishnu Asutosh Dasu
- Main reference** Saeid Tizpaz-Niari, Ashish Kumar, Gang Tan, Ashutosh Trivedi: “Fairness-aware Configuration of Machine Learning Libraries”, in Proc. of the 44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022, pp. 909–920, ACM, 2022.
- URL** <https://doi.org/10.1145/3510003.3510202>
- Main reference** Verva Monjezi, Ashutosh Trivedi, Gang Tan, Saeid Tizpaz-Niari: “Information-Theoretic Testing and Debugging of Fairness Defects in Deep Neural Networks”, in Proc. of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 1571–1582, IEEE, 2023.
- URL** <https://doi.org/10.1109/ICSE48619.2023.00136>
- Main reference** Vishnu Asutosh Dasu, Ashish Kumar, Saeid Tizpaz-Niari, Gang Tan: “NeuFair: Neural Network Fairness Repair with Dropout”, in Proc. of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024, pp. 1541–1553, ACM, 2024.
- URL** <https://doi.org/10.1145/3650212.3680380>

In this lightning talk, I will discuss some of our recent efforts of debugging and fixing fairness issues in machine learning and thoughts about possibly reusing some of the techniques for improving trustworthiness of automated code repair.

### 3.20 More Data or More Domain Knowledge? – For LLM-based Automatic Programming and Repair

*Lin Tan (Purdue University – West Lafayette, US)*

- License** © Creative Commons BY 4.0 International license  
© Lin Tan

Recent techniques use deep learning techniques, including Large Language Models, for automatic programming including automated program repair. An important question is, whether adding more data to train deep learning models or adding domain knowledge to the models is a more promising or effective direction to improve automatic programming. I will discuss existing studies and techniques that answer this question positively or negatively:

#### References

- 1 Xie, Danning and Zhang, Zhuo and Jiang, Nan and Xu, Xiangzhe and Tan, Lin and Zhang, Xiangyu, Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries, CCS 2024
- 2 Nan Jiang and Chengxiao Wang and Kevin Liu and Xiangzhe Xu and Lin Tan and Xiangyu Zhang and Petr Babkin, Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning, ArXiv
- 3 Shanchao Liang and Yiran Hu and Nan Jiang and Lin Tan, Can Language Models Replace Programmers? REPOCOD Says ‘Not Yet’, ArXiv
- 4 Jiang, Nan and Liu, Kevin and Lutellier, Thibaud and Tan, Lin, Impact of Code Language Models on Automated Program Repair, ICSE 2023
- 5 Wu, Yi and Jiang, Nan and Pham, Hung Viet and Lutellier, Thibaud and Davis, Jordan and Tan, Lin and Babkin, Petr and Shah, Sameena, How Effective are Neural Networks for Fixing Security Vulnerabilities?, ISSTA 2023

### 3.21 Neural Code Generation Models with Programming Language Knowledge

*Yingfei Xiong (Peking University, CN)*

License  Creative Commons BY 4.0 International license  
© Yingfei Xiong

Programming requires the awareness of programming language knowledge, such as the syntax, the typing system, and the semantics. Such knowledge is not easily learnable from end-to-end training. In this talk I will introduce a series of work that integrates programming language knowledge into neural models.

### 3.22 Automated Program Repair from Fuzzing Perspective

*Jooyong Yi (Ulsan National Institute of Science and Technology, KR)*

License  Creative Commons BY 4.0 International license  
© Jooyong Yi

APR has been actively researched for the last 15 years, introducing various approaches such as template-based, learning-based, and semantics-based approaches. In particular, the repairability of APR tools has been significantly improved over the years. For example, jGenProg, introduced in 2017, could correctly fix only 2% of the 224 bugs in the Defects4J benchmark, while the latest tool, SRepair, can correctly fix 45% of the 695 bugs in the extended benchmark. This is a remarkable achievement in the field of APR.


While research on repairability should continue, there is another important aspect of APR that has received relatively less attention: APR efficiency. More recent APR tools using template-based or learning-based approaches employ a simplistic method for patch-space exploitation. They simply enumerate the patch candidates in a predefined order based on the suspiciousness scores of the program locations to which the patch candidates are applied. Such an approach is suboptimal because it does not consider the runtime information obtained during the repair process.

In this talk, I present the two recent works we did to improve APR efficiency. We show how the APR scheduling algorithm can be viewed as a multi-armed bandit problem, and how this viewpoint can be used to improve APR efficiency.

The slides for this talk are available at the following link: [https://www.jooyongyi.com/slides/Seminar/Dagstuhl/2024/APR\\_from\\_fuzzing\\_perspective.html](https://www.jooyongyi.com/slides/Seminar/Dagstuhl/2024/APR_from_fuzzing_perspective.html)

### 3.23 Automated Programming in the Era of Large Language Models

*Lingming Zhang (University of Illinois – Urbana-Champaign, US)*

License  Creative Commons BY 4.0 International license  
© Lingming Zhang

In recent years, Large Language Models (LLMs), such as GPT-4 and Claude-3.5, have shown impressive performance in various downstream applications. In this talk, I will discuss the potential impact of modern LLMs on automated programming, including both program repair and synthesis (e.g., AlphaRepair, ChatRepair, and Agentless). In addition, I will also talk about our recent work on rigorous testing/benchmarking of LLMs in automated programming (e.g., EvalPlus and SWE-bench Lite-S).

## 4 Panel discussions

### 4.1 Benchmarks for LLM Code Generation

*Satish Chandra (Google – Mountain View, US), Premkumar T. Devanbu (University of California – Davis, US), Martin Monperrus (KTH Royal Institute of Technology – Stockholm, SE), Gustavo Soares (Microsoft Corporation – Redmond, US), Lin Tan (Purdue University – West Lafayette, US)*

**License** © Creative Commons BY 4.0 International license  
© Satish Chandra, Premkumar T. Devanbu, Martin Monperrus, Gustavo Soares, and Lin Tan

The panel discussion (organized in the format of fishbowl discussion) was led by an academia researcher (Martin Monperrus) and an industrial researcher (Satish Chandra). Existing evaluation benchmarks (e.g., HumanEval) seem rather simple and does not reflect reality. Recent ones try to address this problem by focusing on more realistic tasks (SWEBench, LiveCodeBench, Long Code Arena). However, there are still a lack of benchmark specifically for code maintenance tasks (e.g., refactoring, repo-level changes, code reviewing, code understanding and reasoning, and performance Improvement). An example of such benchmark is CRQBench: A Benchmark of Code Reasoning Questions. The panel also mentioned possible collaboration on “Critical Review on SWE-Bench”. Another possible solution is to have a benchmark track in SE conferences to encourage benchmark curation. The discussion also centered several challenges in benchmarks for LLM Code Generation, including curating good evaluation benchmarks, designing criteria, maintaining and evolving evaluation sets.

### 4.2 LLM-beyond just coding-assistance


*Ahmed E. Hassan (Queen’s University – Kingston, CA), Lin Tan (Purdue University – West Lafayette, US), Shin Hwei Tan (Concordia University – Montreal, CA)*

**License** © Creative Commons BY 4.0 International license  
© Ahmed E. Hassan, Lin Tan, and Shin Hwei Tan

The panel discussion (organized in the format of fishbowl discussion) was led by Ahmed E. Hassan. The discussion focused on discussing the future of automated program Repair (i.e., APR.Next), the challenges, and the opportunities to go from Software Engineering 1.0 to Software Engineering 3.0 where systems based on AI agent have been discussed. Despite the advancement of AI-based techniques, traditional APR techniques still play important roles in: (1) domain specific tasks like Android repair, fixing accessibility issues, (2) improving LLM-generated code by fixing bugs by LLMs using traditional approaches. Other tasks that are worthwhile to explore include: (1) research related to binary repair, and (2) code translation (e.g., C to Rust, translation from COBOL to modern languages).

### 4.3 Obstacles for deploying program repair techniques

*Fernanda Madeiral (VU Amsterdam, NL), Premkumar T. Devanbu (University of California – Davis, US), Gustavo Soares (Microsoft Corporation – Redmond, US)*

**License**  Creative Commons BY 4.0 International license  
© Fernanda Madeiral, Premkumar T. Devanbu, and Gustavo Soares

The panel discussion (organized in according to the format of a fishbowl discussion) was lead by academic researcher (Fernanda Madeiral) and industrial researcher (Gustavo Soares) centered around the obstacles when deploying automated program repair techniques. Challenges mentioned in the discussion include: (1) answering the questions: When (Inner loop, Outer loop)? How (Autonomous agent, Human-in-the-loop)? What (Realistic benchmark)? (2) the need to train developer how to use the technique (3) should write a test that can be reproduce (e.g., the challenge in vulnerability repair lies in writing test for vulnerability. To solve these obstacles, recent work focused on combining the confidence level of a LLM model (e.g., intrinsic probabilities, i.e., probability of a token given previous tokens, per-token log-probs from the model).

## Participants

- Earl T. Barr  
University College London, GB
- Islem Bouzenia  
Universität Stuttgart, DE
- Yuriy Brun  
University of Massachusetts  
Amherst, US
- Cristian Cadar  
Imperial College London, GB
- Celso G. Camilo-Junior  
Federal University of Goiás, BR
- Satish Chandra  
Google – Mountain View, US
- Chunyang Chen  
TU München – Heilbronn, DE
- Tse-Hsun Chen  
Concordia University –  
Montreal, CA
- Zimin Chen  
Deutsche Telekom – Bonn, DE
- Andreea Costea  
National University of  
Singapore, SG
- Premkumar T. Devanbu  
University of California –  
Davis, US
- Alexander Frömmgen  
Google – München, DE
- Ahmed E. Hassan  
Queen's University –  
Kingston, CA
- Sungmin Kang  
KAIST – Daejeon, KR
- Dongsun Kim  
Kyungpook National  
University, KR
- Claire Le Goues  
Carnegie Mellon University –  
Pittsburgh, US
- Yiling Lou  
Fudan University –  
Shanghai, CN
- Fernanda Madeiral  
VU Amsterdam, NL
- Matías Martínez  
UPC Barcelona Tech, ES
- Martin Monperrus  
KTH Royal Institute of  
Technology – Stockholm, SE
- Nikhil Parasaram  
University College London, GB
- Michael Pradel  
Universität Stuttgart, DE
- Nikitha Rao  
Carnegie Mellon University –  
Pittsburgh, US
- Abhik Roychoudhury  
National University of  
Singapore, SG
- André Silva  
KTH Royal Institute of  
Technology – Stockholm, SE
- Gustavo Soares  
Microsoft Corporation –  
Redmond, US
- Gang (Gary) Tan  
Pennsylvania State University –  
University Park, US
- Lin Tan  
Purdue University – West  
Lafayette, US
- Shin Hwei Tan  
Concordia University –  
Montreal, CA
- Yingfei Xiong  
Peking University, CN
- Jinqiu Yang  
Concordia University –  
Montreal, CA
- He Ye  
Carnegie Mellon University –  
Pittsburgh, US
- Jooyong Yi  
Ulsan National Institute of  
Science and Technology, KR
- Jie Zhang  
King's College London, GB
- Lingming Zhang  
University of Illinois –  
Urbana-Champaign, US

