ISSN 2186-7437

NII Shonan Meeting Report

No. 217

Trusted Automated Programming

Corina Păsăreanu
Abhik Roychoudhury
Adish Singla

January 20–23, 2025

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Trusted Automated Programming

Organizers:
Corina Păsăreanu (NASA Ames and CMU, USA)
Abhik Roychoudhury (NUS, Singapore)
Adish Singla (MPI-SWS, Germany)

January 20–23, 2025

1 Introduction and Meeting Overview

The task of programming, both in terms of intent capture as well as in the generation of correct code, has occupied much of the computing profession for the last 50–60 years. There has been significant progress in modeling and system design to support accurate intent capture leading to the growth of formal specifications. However, despite all the progress, software engineers are reluctant to write formal specifications, and for large software systems a formal description of intent is not available leading to tremendous hardship in debugging and fixing errors.

Subsequently, the developments have moved to testing and analysis methods to help develop trustworthy codebases despite the lack of formal capture of intent. These works often go with the goal of achieving higher behavioral coverage as in testing, and often work with simple test-oracles, such as the use of crash-freedom oracles in the widely popular fuzzing techniques. Since the oracles denote the expected behavior of test-cases by assuming trivial oracles, the need for specifying the programmer's intent formally is obviated. Fuzzing methods have witnessed tremendous popularity in the recent decade, with a huge number of vulnerabilities being found in widely used software systems via fuzzing. Nevertheless, the quest to achieve functional correctness in software without having to author voluminous formal specifications remains.

Recent developments in automated code generation from large language models (LLMs) provide us with a fresh perspective in this regard. Since LLM-based code generation allows for programming from natural language specifications, it seems to indicate a promise to achieve the goal of auto-coding. This raises not only the overall question of correctness of automatically generated code, but at what point we can start trusting automatically generated code. In past decades, niche industries have generated code from models, however there is no precedent of automatically generated code from natural language specifications being used widely.

In this Shonan Meeting, we coordinated an exchange among researchers in formal methods, software engineering, and machine learning to address the following questions:

- What is the acceptable evidence for which code from LLMs can be integrated into a software project?

- How do the trust boundaries shift when we integrate automatically generated code instead of manually written code? Are the acceptability criteria likely to be more stringent for automatically generated code?
- Can automatic program repair methods be used to improve the code generated from LLMs? Can repair techniques generate evidence so that the improved code can be accepted into codebases?
- An emerging pattern is counterexample guided synthesis, similar to counterexample guided inductive synthesis (CEGIS), where the LLM is used as a blackbox synthesis engine and one can verify the product of it. Can this approach be used to obtain the required guarantees for the code produced by LLMs?
- Improvement in precision of the LLM models can complement the goal of trustworthiness of automatically generated code. Can these dual criteria lead to a tunable set of parameters in the future, i.e., code from smaller LLMs will need to go through a stringent set of checks and improvement?
- Another approach could be to constrain the output of the LLMs, as fine-tuning LLMs may not be effective in practice; this would be similar to current neuro-symbolic approaches but applied to LLMs. What techniques can be used to effectively constrain the LLMs outputs to obtain desired guarantees?
- How can the advances in LLMs impact programming education and programming teaching of the future? Can it lead to intelligent tutoring systems which interact with both students and LLMs?
- Last but not least, what is the appropriate user study design which will allow us to examine the level of trust developers have in code from LLMs?

We had 28 researchers attending the meeting; the meeting took place over 4 days and comprised talks from researchers, panel sessions, and plenty of time allocated for discussions. In the rest of this report, we provide the list of participants, the schedule of the meeting, the list of talks, a summary of discussion sessions, and the outcome of discussions from the meeting.

2 List of Participants

- Rui Abreu (Faculdade de Engenharia da Universidade do Porto, Portugal)
- Leonhard Applis (National University of Singapore, Singapore)
- Rajeev Alur (University of Pennsylvania, USA)
- Yuriy Brun (University of Massachusetts Amherst, USA)
- Cristian Cadar (Imperial College London, UK)
- Saikat Chakraborty (Microsoft Research, USA)
- Premkumar Devanbu (University of California Davis, USA)
- Sumit Gulwani (Microsoft, USA)
- Wei Le (Iowa State University, USA)
- Claire Le Goues (Carnegie Mellon University, USA)
- Lei Ma (University of Tokyo, Japan)
- Ravi Mangal (Colorado State University, USA)
- Sergey Mechtaev (Peking University, China)
- Yannic Noller (Ruhr University Bochum, Germany)
- Corina Păsăreanu (NASA Ames and Carnegie Mellon University, USA)
- Trung T. Pham (FAA, USA)
- Michael Pradel (University of Stuttgart, Germany)
- Abhik Roychoudhury (National University of Singapore, Singapore)
- Adish Singla (Max Planck Institute for Software Systems, Germany)
- Youcheng Sun (MBZUAI, UAE and Universiry of Manchester, UK)
- Charles Sutton (Google DeepMind, USA)
- Lin Tan (Purdue University, USA)
- Guowei Yang (University of Queensland, Australia)
- Jooyong Yi (UNIST, South Korea)
- Dongmei Zhang (Microsoft Research Asia, China)
- Lingming Zhang (University of Illinois Urbana-Champaign, USA)
- Yuntong Zhang (National University of Singapopre, Singapore)
- Ying Zou (Queen's University, Canada)

3 Meeting Schedule

Check-in Day: January 19 (Sunday)
- 19:00 – 21:00 Welcome banquet

Day1: January 20 (Monday)
- 07:30 – 09:00 Breakfast
- 09:00 – 09:20 Shonan video and introduction by organizers
- 09:20 – 09:45 Participants self-introductions
- 09:45 – 10:30 Talk by Rajeev Alur
- 10:30 – 11:00 Coffee break
- 11:00 – 11:30 Talk by Yuriy Brun
- 11:30 – 12:00 Talk by Claire Le Goues
- 12:00 – 13:30 Lunch
- 13:30 – 14:00 Talk by Michael Pradel
- 14:00 – 14:30 Talk by Yuntong Zhang
- 14:30 – 15:00 Talk by Lingming Zhang
- 15:00 – 15:30 Coffee break
- 15:30 – 16:00 Talk by Wei Le
- 16:00 – 17:00 Panel discussion session
- 17:00 – 18:00 Free time
- 18:00 – 19:30 Dinner

Day2: January 21 (Tuesday)
- 07:30 – 09:00 Breakfast
- 09:00 – 09:45 Talk by Sumit Gulwani
- 09:45 – 10:30 Talk by Trung T. Pham
- 10:30 – 11:00 Coffee break
- 11:00 – 11:30 Talk by Ying Zou
- 11:30 – 12:00 Talk by Saikat Chakraborty
- 12:00 – 12:15 Group photo
- 12:15 – 13:30 Lunch
- 13:30 – 17:00 Excursion along with Japanese tea ceremony
- 17:00 – 18:00 Free time
- 18:00 – 21:00 Banquet

Day3: January 22 (Wednesday)
- 07:30 – 09:00 Breakfast
- 09:00 – 09:30 Free time
- 09:30 – 10:00 Talk by Charles Sutton
- 10:00 – 10:30 Talk by Jooyong Yi
- 10:30 – 11:00 Coffee break
- 11:00 – 11:30 Talk by Lin Tan
- 11:30 – 12:00 Talk by Rui Abreu
- 12:00 – 13:30 Lunch
- 13:30 – 14:00 Talk by Yannic Noller
- 14:00 – 14:30 Talk by Leonhard Applis
- 14:30 – 15:00 Talk by Lei Ma
- 15:00 – 15:30 Coffee break
- 15:30 – 16:00 Talk by Adish Singla

- 16:00 – 17:00 Fishbowl discussion session
- 17:00 – 18:00 Open discussions
- 18:00 – 19:30 Dinner

Day4: January 23 (Thursday)
- 07:30 – 09:00 Breakfast
- 09:00 – 09:30 Free time
- 09:30 – 10:00 Talk by Sergey Mechtaev
- 10:00 – 10:30 Talk by Youcheng Sun
- 10:30 – 11:00 Coffee break
- 11:00 – 11:30 Talk by Ravi Mangal
- 11:30 – 12:00 Summary of discussions by organizers
- 12:00 – 13:30 Lunch

4 Overview of Talks

Neurosymbolic Programming for Trustworthy AI

Rajeev Alur

Neurosymbolic programming combines the complementary worlds of deep learning and symbolic reasoning. It thereby enables more accurate, interpretable, and domain-aware solutions to AI tasks. In this talk, I will give an overview of the state of the art in neurosymbolic programming. I will give examples of how computational problems can be naturally expressed in neurosymbolic frameworks as a composition of a deep neural network followed by a program written in a traditional programming language. The key technical challenge then is to train the neural network based only on end-to-end input-output labels for the composite. I will review some recent learning algorithms addressing this challenge both when the symbolic component is written in a differentiable logic programming language and when the symbolic part is a black-box component. This latter class of algorithms can also support calls to modern LLMs such as GPT-4. I will conclude with an analysis of relative merits of the neurosymbolic approach on benchmarks, potential applications, and remaining challenges. Relevant references: [1, 2]

In Software We Trust

Yuriy Brun

Software is ubiquitous, and trusting it is no longer optional. Unfortunately, modern software has been caught lying, discriminating, and even causing deaths. This talk explores how we can measure trust in software, understand what factors affect trust, and increase software trustworthiness. First, I describe how trust games, an instrument from psychology, can identify what affects users' trust in software. We find, for example, that women prioritize fairness in software more than men do, and that whether text or graphics describe software properties significantly affects users' perception [3]. Second, I introduce Seldonian algorithms that fundamentally re-envision machine learning to produce models that are probabilistically guaranteed to satisfy fairness and safety requirements, even on unseen data [4, 5, 6, 7]. Third, I'll show how cutting-edge natural language processing has enabled fully automatically proving software correctness, the ultimate goal in software trustworthiness. These techniques can synthesize guaranteed-correct proofs (in languages such as Coq and Isabelle) despite machine learning inaccuracies and LLM hallucinations, making formal verification the killer app for LLMs and ensemble learning [8, 9, 10, 11, 12, 13]. Overall, I describe significant progress toward understanding software trust and improving software trustworthiness and lay out the challenges that lay ahead.

Agentic LLM Repair

Claire Le Goues

This talk discusses LLM-based agents for automated program repair, with the idea of using adversarial reasoning to infer the program intent for improved

bug repair.

LLM Agents for Program Repair and Project Setup

Michael Pradel

Large language models (LLMs) are revolutionizing many aspects of software engineering, by providing rich code understanding, code generation, and code editing abilities. Most past work to leverage LLMs in software engineering interacts with the LLM through an a-priori fixed control flow and by providing a fixed set of information to the model. This talk instead presents an agentic approach, where an LLM interacts with a codebase via a set of tools to autonomously perform a specific software engineering task. We present two such approaches: RepairAgent, which addresses the task of program repair, and ExecutionAgent, which addresses the task of automatically installing and setting up a project so that its tests can be executed. Our results show that agentic approaches outperform prior work, providing a new level of LLM-based automation in software engineering. Relevant references: [14, 15]

AutoCodeRover: LLM Agent for Program Improvement

Yuntong Zhang

Recent advancements in Large Language Models (LLMs) have significantly impacted the software development process, enabling developers to leverage LLMs for code generation. However, LLMs still require manual prompting to generate code a specific program locations. We introduce AutoCodeRover, an LLM agent that autonomously handles software maintenance (e.g., program repair) and evolution (e.g., feature addition). AutoCodeRover combines LLMs with advanced code search capabilities to produce program modifications or patches. Rather than treating a software project as a collection of files, AutoCodeRover operates on a program's Abstract Syntax Tree (AST) to retrieve relevant code context. Analysis techniques such as spectrum-based fault localization further refine the code search. In addition, AutoCodeRover extracts explicit natural-language specifications during the search phase, which guide the subsequent code generation process. AutoCodeRover has demonstrated its effectiveness in autonomously resolving GitHub issues and addressing security vulnerabilities identified through fuzz testing on OSS-Fuzz. Related reference: [16].

Automatic Programming in the Age of LLMs

Lingming Zhang

In recent years, Large Language Models (LLMs), such as GPT-4 and Claude-3, have shown impressive performance in various downstream applications, including automatic programming. In this talk, I will discuss the potential impact of modern LLMs on the important problems of test generation (e.g., TitanFuzz) and program repair (e.g., AlphaRepair). Moreover, I will further discuss the recent trend of AI software engineer, and introduce our recent work (Agentless) along this promising direction.

AI for Finding Security Bugs

Wei Le

Vulnerability detection has been a challenging task for program analysis and AI, as it requires an understanding of program semantics. In this talk, I will first illustrate the challenge of this problem by presenting our studies of recent models and state-of-the-art LLMs. I will then highlight our recent work on building different AI models for vulnerability detection, including (1) dataflow inspired models, (2) causality-based approaches, and (3) models built with program traces. The work will shed light on AI for other SE tasks.

Program Synthesis: User Experiences and Neuro-Symbolic Techniques

Sumit Gulwani

Program Synthesis can automate a wide variety of tasks for spreadsheet users (e.g., string transformations, table extraction, advanced data analysis), developers (e.g., debugging, repeated or associated edits), and students (e.g., grading, feedback or hint generation). In this talk, I will demonstrate that user intent can be expressed not only through natural language but also via input-output examples, static and temporal context, and broken artifacts for repair. The most natural way for a user to express intent depends on the task at hand. Additionally, I will advocate for leveraging neuro-symbolic techniques, which combine the power of large language models (LLMs) with logical-reasoning-based symbolic techniques, to build more effective solutions for specific verticals.

Verification of Programming Codes Generated

Trung T. Pham

This talk discusses challenges and methods to certify various components in large-scale, real-world software deployments.

Automated Function Synthesis using LLM Supported Agents

Ying Zou

With the advent of Large Language Models (LLMs), software development has witnessed significant advancements. These models, trained on extensive datasets, have demonstrated remarkable capabilities in performing various coding tasks such as code completion, bug fixing, and code generation. Tools like GitHub Copilot and ChatGPT have gained widespread adoption, with surveys indicating that over 50% of developers now rely on AI-assisted tools in the development process. Studies have also shown that developers using these tools complete tasks significantly faster than those relying solely on traditional methods. Despite these achievements, LLMs face substantial challenges when handling complex coding tasks. While the LLMs can often generate syntactically correct and logically coherent code, their ability to address multi-step programming

problems remains limited. In our work, we propose a novel multi-agent framework that leverages the collaborative strengths of three specialized LLM-based agents for generating an initial code solution, iteratively refining the generated code and dynamically updating fixing strategies to improve the overall performance in code generation. We evaluated the proposed framework using three benchmarks—Live Code Bench, MBPP, and HumanEval—and compared its performance against two baseline approaches: Vanilla LLM and a Self-Repair strategy. The experimental results demonstrate that our framework consistently outperforms both baselines, particularly in Pass@10 and Pass@50 metrics, indicating its ability to iteratively improve generated solutions over multiple repair attempts.

From Intent Formalization to Neural Synthesis: Trustworthy Programming with LLM

Saikat Chakraborty

Integration of artificial intelligence with formal verification techniques is essential to ensure robust, reliable, and efficient systems. With test-driven interactive coding, we demonstrate that guided intent clarification—where developers iteratively refine their requirements through unit tests—can significantly improve code generation accuracy by large language models. This approach addresses the inherent ambiguity in natural language, enabling the transition from informal specifications to executable test cases. Parallel progress in automated specification inference further bridges the gap between developer intent and formal software design. By translating natural language requirements into precise, machine-checkable specifications, these methods empower developers to not only generate code but also rigorously validate it against intended behaviors. This automated formalization process is pivotal for establishing trust in AI-generated code, as it provides a systematic pathway for ensuring correctness. In the domain of proof-oriented programming, the integration of computational content with formal proofs has opened new avenues for constructing verified software. Leveraging extensive datasets of program and proof pairs—augmented by Satisfiability Modulo Theories solvers—we have demonstrated that even a smaller fine-tuned language models can rival larger counterparts in synthesizing both code and correctness proofs. This synergy of AI and formal methods reduces the manual effort typically associated with proving program correctness, thereby streamlining the development process.

LLM Security Agent

Charles Sutton

This talk discusses LLM-based security agents for software vulnerability detection, in particular, for finding input that triggers sanitizer crash.

Security Patch Verification

Jooyong Yi

Timely patching of security vulnerabilities is crucial for maintaining the se-

curity of software systems. With the advancement of automated vulnerability detection techniques such as fuzzing, finding security vulnerabilities is becoming easier. However, the process of patching these vulnerabilities is still largely manual. This technological gap between automated vulnerability detection and manual patching can, contrary to the motivation behind the research on automated vulnerability detection, make the software ecosystem more vulnerable to attacks. To fill this gap, researchers have proposed automated vulnerability repair (AVR) techniques. Given a vulnerable program, AVR techniques automatically generate a patch that fixes the vulnerability. However, most AVR techniques only generate a patch with little or no evidence that the patch is correct. Without evidence for correctness, it is difficult for software developers/maintainers to make an informed decision on whether to apply the patch, which can result in delaying or avoiding the security patch application. In order for an AVR technique to be practically useful, generating a patch is not enough [17]; the automatically generated patch must be verified to ensure its correctness. In this talk, I will present our recent work on patch verification. Our key innovation is to perform bounded verification [18] around the program state that triggers the manifestation of vulnerability.

LLMs for Programming: Benchmark and Domain Knowledge

Lin Tan

This talk discusses Large Language Models (LLMs) for code generation in a broad sense. Here "code" in this context extends beyond source code ([19], [20], and [21]) to include binaries ([22], [23]), HTML and CSS style files ([24]), and LaTeX sources ([25]). We also go beyond the functional correctness of code to consider the security of the code ([26]). For evaluating code generation techniques, we need good benchmarks. RepoCod ([27]) establishes a new code generation benchmark, which contains general code generation tasks, instead of just pull requests from GitHub issues. In addition, RepoCod tasks have repository-level context, whole-function generation, and rigorous validation using test cases. It consists of real-world complex tasks with the longest average canonical solution length (331.6 tokens) and the highest average cyclomatic complexity (9.00). RepoCod includes 314 developer-written test cases per instance for better evaluation. We evaluate ten state-of-the-art LLMs on RepoCod and find that none achieves more than 30% pass@1 on RepoCod, indicating the necessity of building stronger LLMs that can help developers in real-world software development.

Moving Faster and Reducing Risk: Using LLMs in Release Deployment

Rui Abreu

This talk discusses the challenge of determining what should be released in large-scale software development such as at Meta's scale. To address this we developed models to determine the risk of a pull request (diff) causing an outage (aka SEV). We trained the models on historical data and used different types of

gating to predict the riskiness of an outgoing diff. The models were able to capture a significant percentage of SEVs while gating a relatively small percentage of risky diffs. We also compared different models including logistic regression BERT-based models and generative LLMs and found that the generative LLMs performed the best. Related reference: [28]

Simulated Interactive Debugging

Yannic Noller

Debugging software, i.e., the localization of faults and their repair, is a main activity in software engineering. Therefore, effective and efficient debugging is one of the core skills a software engineer must develop. However, the teaching of debugging techniques is usually very limited or only taught in indirect ways, e.g., during software projects. As a result, most Computer Science (CS) students learn debugging only in an ad-hoc and unstructured way. In this talk, we present our approach called Simulated Interactive Debugging that interactively guides students along the debugging process. The guidance aims to empower the students to repair their solutions and have a proper "learning" experience. We envision that such guided debugging techniques can be integrated into programming courses early in the CS education curriculum. Furthermore, we will present the results from an initial evaluation with eight undergraduate CS students. We developed a prototypical IDE-integrated implementation using traditional fault localization techniques and large language models. Students can use features like the automated setting of breakpoints or an interactive chatbot. Based on the responses, we conclude that the participants liked the systematic guidance by the assisted debugger. In particular, they rated the automated setting of breakpoints as the most effective, followed by interactive debugging and chatting. Finally, we will outline our plan to further develop this technology towards agentic workflows for debugging education. Relevant reference: [29]

USEAgent - Unified Software Engineering with Agentic Systems

Leonhard Applis

Agentic Systems are starting to reach software engineers: Their appealing idea is to combine LLMs with the necessary tools, allowing iterative solutions to more complex challenges. Today, most agentic systems specialise in individual tasks like program repair or test execution, resulting in a fragmented landscape of expert tools. This fragmentation might lead to unsustainable solutions - each individual system will need maintenance and introduce barriers through their setup-requirements. What would a better approach look like? We propose a unified software engineering Agent (USEagent), that utilizes other agents and tools to construct a task-specific workflow on the fly. We introduce USEBench, a meta-benchmark combining multiple repository-level SE tasks behind a single API, alongside USEAgent, the latest iteration on AutoCodeRover which targets the diverse tasks of USEBench. USEAgent is capable to self-adjust for program repair, finishing partial fixes, producing test-coverage or generating features

from documentation. We highlight some of the changes to AutoCodeRover, discuss design options and highlight early results.

Towards Understanding the Core of AI Systems in the LLM Era

Lei Ma

In recent years, Large Language Models (LLMs) have significantly advanced artificial intelligence, finding applications across various domains such as software engineering and natural language processing. However, concerns regarding their trustworthiness have emerged, potentially hindering their widespread adoption. Distinct features of LLMs, including self-attention mechanisms, vast model sizes, and autoregressive generation, differentiate them from traditional AI models like Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), posing new challenges for quality analysis. In this presentation, I will discuss our early exploratory study on analyzing the trustworthiness of LLMs and LLM-enabled systems, highlighting the potential to advance this critical area.

GenAI-Powered Educational Technology for Computational Thinking and Programming

Adish Singla

Recent advances in generative AI, in particular deep generative and large language models like ChatGPT, are having transformational effects on the educational landscape. On the one hand, these advances provide unprecedented opportunities to enhance education by creating unique human-machine collaborative systems. On the other hand, the advanced capabilities of these generative AI models have brought unexpected challenges for educators and policymakers worldwide. This talk provides an overview of the research opportunities and challenges in applying generative AI methods for improving computing and programming education. Related references: [30, 31, 32, 33]

Synergizing Program Analysis with Machine Learning

Sergey Mechtaev

This talk discuss techniques to leverage LLMs for automatic program repair in real-world settings. Related references: [34, 35].

Fuzzing Autonomous Systems via Large Language Models

Youcheng Sun

Fuzz testing effectively uncovers software vulnerabilities; however, it faces challenges with Autonomous Systems (AS) due to their vast search spaces and complex state spaces, which reflect the unpredictability and complexity of real-world environments. This paper presents a universal framework aimed at improving the efficiency of fuzz testing for AS. At its core is SAFLITE, a predictive

component that evaluates whether a test case meets predefined safety criteria. By leveraging the large language model (LLM) with information about the test objective and the AS state, SAFLITE assesses the relevance of each test case. We evaluated SAFLITE by instantiating it with various LLMs, including GPT-3.5, Mistral-7B, and Llama2-7B, and integrating it into four fuzz testing tools: PGFuzz, DeepHyperion-UAV, CAMBA, and TUMB. These tools are designed specifically for testing autonomous drone control systems, such as ArduPilot, PX4, and PX4-Avoidance. The experimental results demonstrate that, compared to PGFuzz, SAFLITE increased the likelihood of selecting operations that triggered bug occurrences in each fuzzing iteration by an average of 93.1%. Additionally, after integrating SAFLITE, the ability of DeepHyperion-UAV, CAMBA, and TUMB to generate test cases that caused system violations increased by 234.5%, 33.3%, and 17.8%, respectively. The benchmark for this evaluation was sourced from a UAV Testing Competition. Related reference: [36].

Concept-Based Semantic Analysis of Deep Neural Networks

Ravi Mangal

The analysis of vision-based deep neural networks (DNNs) is highly desirable but very challenging due to the difficulty of expressing formal specifications for vision tasks and the lack of efficient verification procedures. We first describe a logical specification language designed to facilitate writing specifications about vision-based DNNs in terms of high-level, human-understandable concepts. Next, we propose to leverage emerging multimodal, vision-language, foundation models (VLMs) as a lens through which we can reason about vision models. VLMs have been trained on a large body of images accompanied by their textual description, and are thus implicitly aware of high-level, human-understandable concepts describing the images. To encode and formally verify our concept-based specifications, we build a map between the internal representations of a given vision model and a VLM, leading to an efficient verfication procedure for vision models. We demonstrate our techniques on a ResNet-based classifier trained on the RIVAL-10 dataset using CLIP as the multimodal model. We finally conclude by speculating how similar techniques could be used for analysis of code LLMs. Related reference: [37]

5 Overview of Discussion Sessions

Panel Discussion Session: Are Agents the Right Solution or Should We Only Improve LLMs?

Coordinated by Prem Devanbu

The discussion session coordinated by Prem Devanbu looked into the role of agentic AI in automatic programming. Several participants remarked on the fast pace in this research space. Abhik Roychoudhury commented on how agents represent an autonomous workplan and differ significantly from prompts.

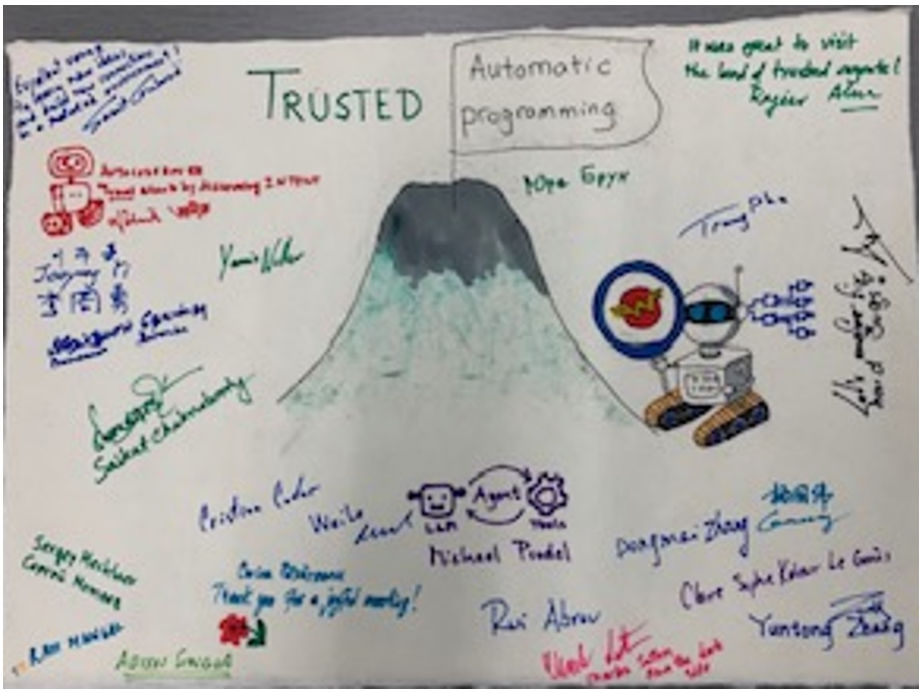
Fishbowl Discussion Session: Challenges of Testing, Verification, and Symbolic Reasoning in the LLM Era

Coordinated by Cristian Cadar

The discussion session coordinated by Cristian Cadar looked into trust issues in automatically generated code. The importance of verification approaches was emphasized by several participants, including Corina Păsăreanu.

6 Outcome of the Discussions

A report co-authored by some of the organizers has been posted on arXiv as an opinion piece [38], and is attracting attention. The final picture jointly drawn and signed by the participants appears below.



References

[1] Ziyang Li, Jiani Huang, Jason Liu, Felix Zhu, Eric Zhao, William Dodds, Neelay Velingker, Rajeev Alur, and Mayur Naik. Relational Programming with Foundational Models. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 10635–10644, 2024.

[2] Alaia Solko-Breslin, Seewon Choi, Ziyang Li, Neelay Velingker, Rajeev Alur, Mayur Naik, and Eric Wong. Data-Efficient Learning with Neural Programs. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[3] Aimen Gaba, Zhanna Kaufman, Jason Cheung, Marie Shvakel, Kyle Wm Hall, Yuriy Brun, and Cindy Xiong Bearfield. My Model is Unfair, Do People Even Care? Visual Design Affects Trust and Perceived Bias in Machine Learning. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 30(1):327–337, 2024.

[4] Philip S. Thomas, Bruno Castro da Silva, Andrew G. Barto, Stephen Giguere, Yuriy Brun, and Emma Brunskill. Preventing Undesirable Behavior of Intelligent Machines. *Science*, 366(6468):999–1004, 2019.

[5] Blossom Metevier, Stephen Giguere, Sarah Brockman, Ari Kobren, Yuriy Brun, Emma Brunskill, and Philip S. Thomas. Offline Contextual Bandits with High Probability Fairness Guarantees. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, pages 14893–14904, 2019.

[6] Stephen Giguere, Blossom Metevier, Yuriy Brun, Bruno Castro da Silva, Philip S. Thomas, and Scott Niekum. Fairness Guarantees under Demographic Shift. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2022.

[7] Austin Hoag, James E. Kostas, Bruno Castro da Silva, Philip S. Thomas, and Yuriy Brun. Seldonian Toolkit: Building Software with Safe and Fair Machine Learning. In *Proceedings of the International Conference on Software Engineering (ICSE) Demo Track*, pages 107–111, 2023.

[8] Emily First, Yuriy Brun, and Arjun Guha. TacTok: Semantics-Aware Proof Synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue*, 4:231:1–231:31, 2020.

[9] Emily First and Yuriy Brun. Diversity-Driven Automated Formal Verification. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 749–761, 2022.

[10] Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 45(2):12:1–12:30, 2023.

[11] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1229–1241, 2023.

[12] Robert Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2025.

[13] Alex Sanchez-Stern, Abhishek Varghese, Zhanna Kaufman, Dylan Zhang, Talia Ringer, and Yuriy Brun. QEDCartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2025.

[14] Islem Bouzenia, Premkumar T. Devanbu, and Michael Pradel. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *CoRR*, abs/2403.17134, 2024.

[15] Islem Bouzenia and Michael Pradel. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *CoRR*, abs/2412.10133, 2024.

[16] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISTA)*, pages 1592–1604, 2024.

[17] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Concolic Program Repair. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 390–405, 2021.

[18] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 157–166, 2006.

[19] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1251–1263, 2023.

[20] Jinhao Dong, Yiling Lou, Dan Hao, and Lin Tan. Revisiting Learning-based Commit Message Generation. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 794–805, 2023.

[21] Ruixin Wang, Zhongkai Zhao, Le Fang, Nan Jiang, Yiling Lou, Lin Tan, and Tianyi Zhang. Show Me Why It's Correct: Saving 1/3 of Debugging Time in Program Repair with Interactive Runtime Comparison. *Proceedings of ACM Programming Languages*, (OOPSLA), 2025.

[22] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proceedings of the SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[23] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[24] Shanchao Liang, Nan Jiang, Shangshu Qian, and Lin Tan. WAFFLE: Multi-Modal Model for Automated Front-End Development. *CoRR*, abs/2410.18362, 2024.

[25] Nan Jiang, Shanchao Liang, Chengxiao Wang, Jiannan Wang, and Lin Tan. LATTE: Improving Latex Recognition for Tables and Formulae with Iterative Refinement. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2025.

[26] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1282–1294, 2023.

[27] Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. Can Language Models Replace Programmers? REPOCOD Says 'Not Yet'. *CoRR*, abs/2410.21647, 2024.

[28] Rui Abreu, Vijayaraghavan Murali, Peter C Rigby, Chandra Sekhar Maddila, Weiyan Sun, Jun Ge, Kaavya Chinniah, Audris Mockus, Megh Mehta, and Nachiappan Nagappan. Moving Faster and Reducing Risk: Using LLMs in Release Deployment. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2025.

[29] Yannic Noller, Erick Chandra, Srinidhi HC, Kenny T. W. Choo, Cyrille Jégourel, Oka Kurniawan, and Christopher M. Poskitt. Simulated Interactive Debugging. *CoRR*, abs/2501.09694, 2025.

[30] Tung Phung, Victor-Alexandru Padurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generative AI for Programming Education: Benchmarking Chatgpt, Gpt-4, and Human Tutors. In *Proceedings of the Conference on International Computing Education Research (ICER) - Volume 2*, 2023.

[31] Tung Phung, Victor-Alexandru Padurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. Automating Human Tutor-Style Programming Feedback: Leveraging GPT-4 Tutor Model for Hint Generation and GPT-3.5 Student Model for Hint Validation. In *Proceedings of the Learning Analytics and Knowledge Conference (LAK)*, pages 12–23, 2024.

[32] Nachiket Kotalwar, Alkis Gotovos, and Adish Singla. Hints-In-Browser: Benchmarking Language Models for Programming Feedback Generation. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[33] Victor-Alexandru Pădurean, Paul Denny, and Adish Singla. BugSpotter: Automated Generation of Code Debugging Exercises. In *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE)*, 2025.

[34] Nikhil Parasaram, Huijie Yan, Boyu Yang, Zineb Flahy, Abriele Qudsi, Damian Ziaber, Earl T. Barr, and Sergey Mechtaev. The Fact Selection Problem in LLM-Based Program Repair. In *Proceedings of the International Conference on Software Engineering (ICSE)*, 2025.

[35] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. User-Centric Deployment of Automated Program Repair at Bloomberg. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 81–91, 2024.

[36] Taohong Zhu, Adrians Skapars, Fardeen Mackenzie, Declan Kehoe, William Newton, Suzanne M. Embury, and Youcheng Sun. SAFLITE: Fuzzing Autonomous Systems via Large Language Models. *CoRR*, abs/2412.18727, 2024.

[37] Ravi Mangal, Nina Narodytska, Divya Gopinath, Boyue Caroline Hu, Anirban Roy, Susmit Jha, and Corina S Păsăreanu. Concept-Based Analysis of Neural Networks via Vision-Language Models. In *International Symposium on AI Verification (SAIV)*, pages 49–77. Springer, 2024.

[38] Abhik Roychoudhury, Corina Păsăreanu, Michael Pradel, and Baishakhi Ray. Agentic AI Software Engineer: Programming with Trust. *CoRR*, abs/2502.13767, 2025.