AI Assisted Programming
(AISoLA 2025 Track Introduction)

Wolfgang Ahrendt[1], Bernhard K. Aichernig[2], and Klaus Havelund[3⋆]

[1] Chalmers University of Technology and University of Gothenburg, Sweden
ahrendt@chalmers.se
[2] Inst. for Formal Models and Verification, Johannes Kepler University Linz, Austria
bernhard.aichernig@jku.at
[3] NASA Jet Propulsion Laboratory, California Inst. of Technology, USA
klaus.havelund@jpl.nasa.gov

Abstract. This is an introduction to the track 'AI Assisted Programming' (AIAP), organized at the third instance of the AISoLA conference during the period November 1 - 5, 2025. AISoLA as a whole aims to study opportunities and risks of late advances of AI. The motivation behind the AIAP track in particular, which also takes place the third time, is the emerging use of large language models for the construction and analysis of software artifacts. An overview of the track presentations is provided.

1 Introduction

Neural program synthesis, using Large Language Models (LLMs) which are trained on open source code, have quickly become a popular addition to the software developer's toolbox. LLMs like, for instance, OpenAI's ChatGPT [9], Anthropic's Claude [10], Google's Gemini [11], xAI's Grok [12], Meta's LLaMA [13]; and various LLM enhanced IDEs such as Copilot [14], Cursor [15], and Windsurf [16], can generate code in many different programming languages from natural language requirements. This opens up for fascinating new perspectives, such as increased productivity and accessibility of programming also for non-experts. However, neural systems do not come with guarantees of producing correct, safe, or secure code. They produce the most probable output, based on the training data, and there are countless examples of coherent but erroneous results. Even alert users fall victim to automation bias: the well studied tendency of humans to be over-reliant on computer generated suggestions. The area of software development is no exception to this automation bias.

The track *AI Assisted Programming* at AISoLA 2025 is the third of its kind, after the first instance in 2023 [2] and the second instance in 2024 [1]. It is devoted to discussions and exchange of ideas on questions like: What are the

⋆ The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

capabilities of this technology when it comes to software development? What are the limitations? What are the challenges and research areas that need to be addressed? How can we facilitate the rising power of code co-piloting while achieving a high level of correctness, safety, and security? What does the future look like? How should these developments impact future approaches and technologies in software development and quality assurance? What is the role of models, tests, specification, verification, and documentation in conjunction with code co-piloting? Can quality assurance methods and technologies themselves profit from the new power of LLMs?

2 Contributions

The above questions are taken up by the participants of the track in eleven talks. Six talks [3,4,5,6,7,8] are associated with regular papers. The remaining five talks do not have associated papers in the proceedings. Presenters have been offered to publish regular papers in subsequent post-conference proceedings.

2.1 Talks with Papers in the Proceedings

Khashayar Etemadi, Marjan Sirjani, Mahshid Helali Moghadam, Per Strandberg, and Paul Pettersson (*LLM-based Property-based Test Generation for Guardrailing Cyber-Physical Systems* [3]) propose an automated method for guardrailing cyber-physical systems (CPSs) using property-based tests (PBTs) generated by LLMs. Their approach uses an LLM to extract system properties from CPS code and documentation, then generates PBTs to verify these properties both at design time (pre-deployment testing) and at run time (monitoring to prevent unsafe states). They implement the method in a tool called ChekProp and evaluate it on preliminary case studies, measuring the generated PBTs' relevance, executability, and effectiveness in covering input space partitions. Results indicate that LLM-generated PBTs offer a promising direction for CPS safety assurance.

Moez Ben Hajhmida and Edward A. Lee (*RAG and Agentic Assistant: A Combined Approach* [4]) present a hybrid approach for translating Lingua Franca (LF) programs that use the C target into equivalent LF programs using the Python target. LF is a coordination language for reactor-based architectures, where individual actors are programmed in popular programming languages including C and Python. Converting 150 C regression tests into according Python versions, the method combines Retrieval-Augmented Generation (RAG), which retrieves similar LF-Python examples to guide code LLMs, with an agentic AI assistant in the Cursor IDE to automate syntax correction, refactoring, and code standardization. The results show that RAG greatly improves small-model performance, and the assistant further increases the proportion of syntactically correct and executable translations.

Niclas Hertzberg, Merlijn Sevenhuijsen, Liv Kåreborn, and Anna Lokrantz (*CASP: An Evaluation Dataset for Formal Verification of C code* [5]) present CASP, a benchmark dataset for evaluating LLMs and other automated tools on

the generation and verification of C code against formal specifications. CASP is built from The Stack v1 and v2 — large, permissively licensed source code repositories from the BigCode project — by extracting self-contained C functions annotated with ANSI/ISO C Specification Language (ACSL), verifying them with the Frama-C framework, and repairing faulty files using automated and manual methods. The resulting 506 function–specification pairs enable reproducible benchmarking for tasks such as generating code from specifications, deriving specifications from code, and repairing non-verifying pairs, supporting research toward more reliable, formally verified software systems.

Lennart Landt, Martin Leucker, and Carsten Burchardt (*A Voice-Enabled Query Framework for Systems Engineering Artefacts* [6]) propose a voice-enabled AI framework to improve comprehension and exploration of Model-Based Systems Engineering (MBSE) models, particularly for newcomers and interdisciplinary teams. While MBSE, often implemented in SysML, supports complex, collaborative design, it has a steep learning curve. The presented framework employs AI avatars representing different engineering roles, enabling natural-language voice queries about system artifacts. A processing pipeline converts MBSE model data into a machine-readable form for LLMs, which generate contextual, role-specific responses. The prototype supports a multi-turn dialogue, helping users to navigate and interpret models, fostering collaboration, and lowering barriers to effective MBSE adoption.

Julian Müller, Thies de Graaff, and Eike Möhlmann (*Integrating LLMs with QC-OpenDRIVE: Ensuring Normative Correctness in Autonomous Driving Scenarios* [7]) investigate integrating LLMs with QC-OpenDRIVE to generate OpenDRIVE road network files that are both syntactically valid and compliant with domain rules for autonomous driving scenario validation. While LLMs can easily produce diverse road layouts, they often break normative requirements such as a rule which mandates geometric continuity between connected roads — their endpoints, tangents, and curvature must align seamlessly. The proposed approach adds a feedback loop: QC-OpenDRIVE validates LLM output, flags semantic and normative errors, and the LLM iteratively corrects them. Combining this loop with Retrieval-Augmented Generation or reasoning steps yields valid results, demonstrating the value of domain-specific validation.

Amer Tahat, Isaac Amundson, David Hardin, and Darren Cofer (*AGREE-Dog Copilot: A Neuro-Symbolic Approach to Enhanced Model-Based Systems Engineering* [8]) present AGREE-Dog, an open-source generative AI copilot for the AGREE compositional reasoning tool, aimed at making model-checking counterexamples easier to understand and resolve. Large, tabular counterexamples can overwhelm engineers, especially newcomers. AGREE-Dog, integrated into the OSATE IDE, uses LLMs to explain violations, suggest repairs, and automate DevOps and ProofOps steps — such as re-running analyses and managing updated models — so users can iterate quickly. A context-selection and memory system tracks evolving artifacts and past interactions, while new structural and temporal metrics measure how much manual input is required. In 13 fault-

injection scenarios, AGREE-Dog achieved rapid, accurate counterexample repair with minimal human effort.

2.2 Talks without Papers in the Proceedings

Lenz Belzner, Thomas Gabor, and Martin Wirsing (*AI Engineering vs. Vibe-Coding: a Strategic Look at AI-Assisted Software Engineering*) offer a strategic perspective on the rise of AI-assisted software engineering, contrasting its transformative potential with the risks of "vibe coding" — code that appears functional but lacks robust architecture, documentation, and maintainability. They note that AI agents can now assist across the software lifecycle, from requirements analysis and design to automated code and test generation, promising gains in productivity, speed, and complexity management. However, without deliberate safeguards, AI-generated systems risk technical debt, security vulnerabilities, and reduced long-term stability. The talk explores how to harness AI's advantages while preserving quality, testability, and security, ensuring AI becomes a tool for strategic excellence.

Itay Cohen, Klaus Havelund, and Doron Peled (*Synthesizing Runtime Verification Monitors with LLMs*) investigate using LLMs to synthesize runtime verification (RV) monitors directly from natural-language specifications, extending beyond narrowly defined formalisms like linear temporal logic. Their tool engages in structured interaction with an LLM to interpret rich, often ambiguous constructs from design documents, generating multiple plausible interpretations and refining them with user feedback. This process builds a reusable library of temporal constructs that can be automatically translated into monitor code. By mediating LLM interactions through the tool, the approach enhances expressive power, better aligns with original intent, and improves the trustworthiness and reliability of generated monitors for verification purposes.

Lucas Cordeiro (*AI-Assisted Formal Verification: Towards Fast, Accessible, and Rigorous Software Verification*) presents four AI–formal verification integrations. One approach combines large language models with bounded model checking to automatically derive formal properties from natural-language requirements, demonstrated on industrial cyber-physical systems and supporting richer logical expressiveness while reducing false positives. Another uses a lightweight AI model for real-time classification of potential vulnerabilities in C, C++, Java, Python, Kotlin, and Solidity. A third introduces an autonomous repair framework that detects flaws such as buffer overflows and pointer dereference errors in C/C++ code, generates fixes, and validates them formally. Finally, a loop summarization technique mitigates state-space explosion in verifying programs with complex or nested loops.

João F. Ferreira (*Techniques and Experiments in Retrieval-Augmented Neural Theorem Proving*) explores the potential of large language models for automated theorem proving, tracing developments from early retrieval-augmented approaches to recent reinforcement learning experiments. The talk first describes Rango, a system that dynamically adapts to the current proof state by retrieving and applying relevant lemmas and prior proofs. The talk then presents ongoing

work using Group Relative Policy Optimization (GRPO) to train models for improved reasoning performance. The talk addresses the challenges of integrating retrieval, learning, and logical inference, reports observed performance gains, and outlines future research directions toward combining these components to advance neural theorem proving capabilities.

Jie He, Vincent Theo Willem Kenbeek, Zhantao Yang, Meixun Qu, Ezio Bartocci, Dejan Ničković, and Radu Grosu (*Explaining Timing Diagrams with LLMs*) present a multimodal approach to assist engineers in understanding complex timing diagrams (TDs) originating from third-party sources, commonly encountered during hardware design and verification. This approach offers an interactive visual question-answering interface, enabling users to upload TDs and ask targeted questions about signal behavior, timing constraints, and protocol correctness.

Alexandra Mendes (*LLM-Assisted Program Correctness: Generating Lemmas, Assertions, and Repairs in Dafny*) examines how large language models can assist in overcoming common bottlenecks in formal verification with Dafny. While Dafny provides strong correctness guarantees, verification often requires developers to supply helper assertions, loop invariants, and lemmas, a process that is both time-consuming and error-prone. The talk presents two applications of LLMs: generating missing assertions and lemmas, and performing specification-guided automated program repair. It also discusses observed strengths and limitations of LLMs in this context, emphasizing how combining symbolic reasoning with model-generated suggestions can improve efficiency, reduce manual effort, and make formal verification more accessible to a broader range of developers.

Jonas Schiffl, Samuel Teuber, and Bernhard Beckert (*Formally Verified LLM Program Synthesis for Solidity Smart Contracts*) present an approach to automatically synthesize formally verified Solidity smart contracts using LLMs within the Scar model-driven verification-based development process. In Scar, developers first create an abstract model with security and correctness properties, from which a formally specified code skeleton is generated. Traditionally, developers manually implement this skeleton and verify it against the specification. Here, an LLM generates the implementation directly from the specification, followed by automated formal verification using Certora and solc-verify. If verification succeeds, the code is guaranteed correct. The method is evaluated on multiple use cases, comparing both verification tools and reporting practical insights.

Shivkumar Shivaji, Natalia Lobakhina, Lucas Cordeiro, and Klaus Havelund (*LLM-Assisted Program Translation and Bounded Model Checking for Formal Verification of Python Code*) present a framework for verifying Python programs by combining large language model–based program translation with bounded model checking. An RLHF-enhanced LLM translates Python code into semantically equivalent C, which is then analyzed using the ESBMC bounded model checker to verify safety properties through intelligent path exploration. This approach bridges high-level Python development with rigorous formal verification. Supporting formal verification of Python allows the popular Python language

to be used as a modeling language instead of traditional formal specification languages, which usually have steep learning curves and have limited expressiveness. This is demonstrated on a model of an autonomous Lunar rover control system.

Cheng Wang, Florian Lorber, Edi Muškardin, and Bernhard Aichernig (*Formal Verification of AI-based Code Generation in Model-Driven Development*) propose a formal evaluation method for LLM code generation, using finite-state machines as ground truth specifications. These models are automatically translated into natural language descriptions and provided to an LLM, which generates Python programs intended to match the original behavior. The generated programs are then analyzed with active automata learning (using the AALpy automata learning library) to infer their input–output behavior and compare it against the ground truth, producing a similarity score. The method also supports iterative repair of faulty code using counterexamples. Initial experiments with four popular LLMs on randomly generated Mealy machines reveal differences in accuracy and robustness.

3 Conclusion

The presentations in this track cover the use of LLMs in the context of all phases of software development, including requirements, designs, coding, testing and verification. This includes such topics as LLM support for specification generation, test case generation, runtime verification, formal verification, auomated repair, translation of high-level design models and specifications to code, translation between programming languages, human comprehension of models, and benchmarks. This covers an interesting spectrum of AI assisted programming. We hope that this track, with its talks, discussions, and papers, contributes to a future of AI assisted programming which exploits the strengths of arising AI technologies while mitigating the corresponding risks. We are convinced that many communities within computing have a lot to contribute to such a development, and look forward to future initiatives and contributions towards this aim.

References

1. W. Ahrendt, B. Aichernig, and K. Havelund. AI assisted programming. In B. Steffen, editor, *AISoLA 2024 - Bridging the Gap Between AI and Reality*, volume 15217 of *LNCS*, pages 101–106. Springer International Publishing, 2025.
2. W. Ahrendt and K. Havelund. AI assisted programming. In B. Steffen, editor, *AISoLA 2023 - Bridging the Gap Between AI and Reality*, volume 14380 of *LNCS*, pages 351–354. Springer International Publishing, 2024.
3. K. Etemadi, M. Sirjani, M. H. Moghadam, P. Strandberg, and P. Pettersson. LLM-based property-based test generation for guardrailing cyber-physical systems. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].

4. M. B. Hajhmida and E. A. Lee. RAG and agentic assistant: A combined approach. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].
5. N. Hertzberg, M. Sevenhuijsen, L. Kåreborn, and A. Lokrantz. CASP: An evaluation dataset for formal verification of c code. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].
6. L. Landt, M. Leucker, and C. Burchardt. A voice-enabled query framework for systems engineering artefacts. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].
7. J. Müller, T. de Graaff, and E. Möhlmann. Integrating LLMs with QC-OpenDRIVE: Ensuring normative correctness in autonomous driving scenarios. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].
8. A. Tahat, I. Amundson, D. Hardin, and D. Cofer. AGREE-Dog Copilot: A neuro-symbolic approach to enhanced model-based systems engineering. In *Proc. of AISoLA 2025 - Bridging the Gap Between AI and Reality. Track: AI Assisted Programming*, LNCS. Springer International Publishing, 2025. [in this volume].
9. ChatGPT LLMs (OpenAI). https://chat.openai.com.
10. Claude LLMs (Anthropic). https://claude.ai.
11. Gemini LLMs (Google). https://gemini.google.com.
12. Grok LLMs (xAI). https://grok.com.
13. LLaMA LLMs (Meta). https://www.llama.com.
14. Copilot IDE. https://copilot.github.com.
15. Cursor IDE. https://cursor.com.
16. Windsurf IDE. https://windsurf.com.